Lehrstuhl für Effiziente Algorithmen
des Instituts für Informatik
der Technischen Universität München

Randomised Dynamic Load Balancing

Tom Friedetzky

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Dr. h.c. W. Brauer

Prüfer der Dissertation:

1. Univ.-Prof. Dr. E.W. Mayr
2. Univ.-Prof. Dr. F. Meyer auf der Heide,
   Universität Gesamthochschule Paderborn

Die Dissertation wurde am 28.06.2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16.08.2002 angenommen.

Acknowledgements

There are many people I would like to express my thanks to. My first big "thank you" goes to my mother. For everything. The second one goes to my brother, who never ceased to remind me that it indeed *is* a bit odd that serious people spend some considerable time and effort in analysing the process of randomly throwing balls into bins. Can't say I blame him...

Further thanks go to two persons I have had the luck to meet during my time at universities. In chronological order: I am deeply indebted to the advisor of my master's thesis, Prof. Dr. Friedhelm Meyer auf der Heide. Blame him for waking the love for theoretical computer science in me. I of course also owe thanks to the advisor of this PhD thesis, Prof. Dr. Ernst W. Mayr, for always letting me freely choose my way (although he insisted on me performing the Bavarian rite of quickly emptying a bottle of good Bavarian beer into a glass without setting down — a thing I'm not too good at, I'm afraid).

Further thanks go to my colleagues at the *Chair for Efficient Algorithms* for providing such an extremely pleasant working environment. I know I almost never joined you for lunch... but that was for different reasons.

Special thanks go to my former office mate, Thomas "Mr. Counter Example" Erlebach. I miss the most inspiring discussions about death stars, data structures, the `vi` text editor, and heavy metal music.

Finally, the most important thanks. Petra, thank you for always being there. I couldn't imagine life without you.

Tom Friedetzky

Abstract

Load balancing is about distributing work load (jobs, tasks, processes, etc.) among a set of processing facilities (processors, workstations, servers, etc.), such that, usually, this load is more or less evenly distributed.

In this work we introduce and investigate the performance of three randomised load balancing algorithms for dynamic settings, that is, we are interested mostly in the long term behaviour of the algorithms, and here especially in deriving an upper bound on the maximum load of any server at any point of time.

We assume two fundamentally different load generation schemes. First, we analyse the algorithms given a stochastical scheme, i.e., the generation and consumption of load obeys some probabilistic distribution. Second, we introduce an adversarial scheme, where we assume that the generation and consumption of load is controlled by an adversary, who deliberately tries to produce a load distribution as uneven as possible. Two of our algorithms are for the stochastical generation scheme, and the third one is for the adversarial scheme.

After rigorously analysing the performance of the algorithms, we present some simulation results that indicate that under certain conditions our algorithms very are well-behaved in practice, too. Since the algorithms are somewhat theoretical in nature, we also briefly discuss how to modify them and tune certain parameters in order to make them even more applicable for practical problems.

Contents

1 Introduction **1**

1.1 Classification of Load Balancing Approaches 2

1.2 Our Model of Computation 5

1.3 Previous Work 8

1.3.1 Balls Into Bins Approaches 8

1.3.2 Other Approaches 10

1.3.3 Practical Work 12

1.4 General Survey of our Algorithms 12

1.5 Load Generation and Service Models 13

1.5.1 The Stochastical Model 13

1.5.2 The Adversarial Model 15

1.6 New Results 17

1.6.1 The Stochastical Model 17

1.6.2 The Adversarial Model 21

2 The Collision Protocol **23**

3 The Balancing Algorithm ALGSTOCHMULTICOLL **29**

3.1 The Algorithm 30

3.2 The Analysis 33

3.2.1 The Non-Balancing System 34

3.2.2 From Non-Balancing to Balancing Systems 39
3.2.3 A Single Phase 42
3.2.4 Proving the Main Theorem 49
3.2.5 Other Generation Models 52
3.2.6 Recovery Properties 56

4 The Balancing Algorithm AlgStochSingleColl 59

4.1 The Algorithm 60
4.2 The Analysis 63
4.2.1 A Single Phase 64
4.2.2 Proving the Main Theorem 74
4.2.3 Other Generation Models 75

5 The Balancing Algorithm AlgAdv 77

5.1 The Algorithm 77
5.2 The Analysis 82
5.2.1 The Load Estimation Phase 83
5.2.2 The Assignment Sub-Phase 87
5.2.3 From Phase To Phase 92
5.2.4 Expected Behaviour 95
5.2.5 Worst Case Recovery 96

6 Simulations 105

6.1 Technical Difficulties 105
6.2 Basic Simulation Rules 107
6.3 The Collision Protocol 108
6.4 The Balancing Algorithm AlgStochMultiColl 108
6.5 The Balancing Algorithm AlgStochSingleColl 111
6.6 The Balancing Algorithm AlgAdv 120

7 Conclusions and Outlook **125**

Bibliography **127**

CHAPTER 1

Introduction

During the last couple of years, parallel computers have been used to perform certain computations where execution on a standard one-processor machine would have been much too time consuming, both in the academic research world and in commercial companies. One can, for instance, think of production scheduling needed to be done by companies, weather forecast, simulations in physics, and so on. One critical issue with parallel computers is load balancing.

In general, load balancing is the process of distributing load units (tasks, jobs) among a set of processing facilities (processors in a massive parallel system, or workstations in a workstation cluster, for instance) in order to achieve certain objectives. These objectives can (and do) vary heavily from application to application, and from system to system.

One can, for instance, think of a university's workstation pool, where some workstations are completely idle (no one is sitting in front of them), some have a small work load (students in front of them are just reading their email or things like that), and some have a high work load (students start certain processing time intensive simulations, run TEX on their thesis, and enjoy a game of Doom now and then). Additionally, on top of all this, the staff utilises the machines to perform their own computations. Clearly, an even distribution of work load among the machines would be a highly desirable objective. This could be achieved if newly generated tasks would not necessarily be executed on the machines where they are generated on, but under certain circumstances on some other, less utilised machine – the

very idea of load balancing. Obviously, there are a couple of things to take care of, like how to actually find a less loaded machine, to ensure that one doesn't pay with too large a network communication overhead due to the transfer of tasks, and many many more.

In practice, one would choose the load balancing algorithm depending on what kind of application needs to be run, on what kind of system this application runs, and on what parameters are subject to optimisation. This directly leads to the following small classification of load balancing approaches.

SECTION 1.1

Classification of Load Balancing Approaches

In this section we give a very brief overview on some distinctive properties of different load balancing approaches and problems, respectively. This overview is by no means to be meant even remotely complete, it just covers what is necessary and important to classify the algorithms presented here. For a more complete classification, see for instance [SG97].

Communication restrictions. Some load balancing algorithms are restricted to neighbour-to-neighbour communication, where the neighbour structure is given by the topology of the network, whereas others allow communication between any set of servers (of course, in case of a connected network, general communication always can be "simulated" by appropriate routing strategies). Furthermore, some approaches allow a server to communicate with a bounded number of other servers at a time step, whereas others do not place such a restriction.

Global and local approaches. In global algorithms there is some centralised instance that decides when and/or where to transfer load. This instance has a view over the complete system. On the other hand, in local algorithms a processor/machine makes its own decisions. This distinction must not be confused with how much information actually is needed to make these decisions. Local load balancing algorithms can very well also

employ full knowledge of the current distribution of the load. A server could decide to gather this knowledge one way or the other, and then make its decision based on this knowledge.

Static and dynamic problems. The major difference between static and dynamic load balancing problems is that in the case of static problems one has a fixed set of tasks that have to be executed, whereas in the dynamic case new tasks are generated and executed as time passes. An example for the static case is the calculation of the roots of some polynomial. This problem can be split into some (more or less) independent sub-problems which then can be executed on different machines. Most important is that after solving the sub-problems one has solved the overall problem and there are no new problems coming up (unless a new polynomial has to be processed, that is). An example for the dynamic case is the already mentioned workstation pool. Students come and go as time passes and one cannot simply say "hey, it's 12 pm, so that's it" — students just keep on coming and going and issuing tasks to be serviced.

Preselected location of tasks. In some cases tasks come with some initial placement, and it is not strictly necessary to move them to some other machine prior to their service. Note that it may well be sensible (in order to to achieve one or another objective, say, an even distribution of the working load) — but it is not necessary at all. In other cases tasks do not have any initial placement. In this case, one can think of a computation server consisting of several server machines, and a couple of user terminals connected to this server cluster via one or more gateways. Now, if a task is issued on a user terminal, it arrives at one of the gateways and from there it is moved to one of the server machines, where it actually is executed. For example, in a WWW based database queries arrive at the web server which, in turn, forwards them to machines which are evaluating them. This approach is fundamentally different from, say, a simple workstation cluster as described above, where tasks are generated on specific workstations (which is their preselected location) and, if nothing happens which causes a transport to some other machine, they are executed on the machines where they have been generated on.

What to optimise? As mentioned before, there are quite some parameters that can be subject to optimisation by load balancing approaches. Frequently they are given due to certain resource bottlenecks. If, for instance, processing time is crucial, then minimising the number of idle servers or an even distribution of the work load are the things to have in mind. If (physical) memory is expensive, then minimisation of queue lengths is highly desirable.

Dependent and independent tasks. Some applications spawn tasks in a way (or can only be split into tasks such that) these tasks are dependent — be it that they need to exchange information or that there is some precedence structure among them, that is task X can run if and only if task Y has already finished because it needs Y's result first. Obviously, a load balancing algorithm has to take possible dependencies into account. If it would not do so and distribute tasks regardless of these dependencies, the communication overhead could become enormously, even up to the case where computation on a single-processor machine would not have been much slower than what has been achieved on the multi-processor machine.

Randomised and deterministic algorithms. Of course, the separation into these two classes is important not only for load balancing algorithms. Generally, one would like to devise deterministic algorithms unless they prove to be highly inefficient. Generally, randomisation comes into play when the problem is irregular, or to handle known worst case instances of certain problems.

Load generation. Basically, this describes how new tasks are generated. This is meaningful only for dynamic problems, since in static ones no new tasks are generated at all as time passes. There are quite some imaginable load generation schemes: strictly deterministic (the generation distribution is not only predictable but fixed and known), stochastical (the load generation obeys some stochastical distribution), or even adversarial (some adversary tries to fool the algorithm into making wrong decisions and being being sub-optimal).

Load servicing. Closely related to load generation is load servicing. More precisely, load servicing describes how long certain tasks need to be serviced. Again, the classification from above still is valid; the service time can be distributed deterministically (each task has a service time of, say, constant one), or distributed stochastically, or be controlled by some adversary.

In Section 1.2 we describe how our approaches fit into this classification and introduce our model in detail. But at first, we give an overview on some selected known results in the area of load balancing.

SECTION 1.2

Our Model of Computation

Before we come to introduce the results presented in this work, we first describe our model of computation, following the classification of Section 1.1.

We assume a system of n synchronously working servers, where any two servers are able to communicate, that is, the network connecting the servers can be seen as an undirected connected graph $G = (V, E)$. A node $v \in V$ represents a server, and an edge $e \in E$ represents a communication link connecting two servers. We do make no further assumptions regarding the topology of the network (of the graph) other than that it is connected. We make, however, the restriction that at any time step a server may communicate with at most a constant number of other servers. In order to focus on the load balancing mechanisms we do not directly take into account costs for routing of messages or transfer of tasks, respectively. Clearly, if two servers not directly connected want to communicate, the messages have to be routed between them somehow. We do not specify how this is done. We make, however, one assumption. Since most reasonable and realisable networks have a diameter of $\mathcal{O}(\text{polylog}(n))$, and consequently routing of a message can take as long (communication among two servers), we allow for $\mathcal{O}(\text{polylog}(n))$ steps of internal computation for any step of communication. This seems to be a quite reasonable assumption.

In our model, time consists of consecutive discrete steps. In such a step,

besides the normal load generation and service, servers are able to perform additional work which is needed entirely for the load balancing. One can think of some sort of background processes. This additional balancing work consists of communication (under the restrictions as described above), and local computation, which may be needed to make certain decisions necessary for the balancing algorithm (as, for instance, to decide if, how many, and where to pass away tasks).

The reader should keep in mind that if in the network in question communication via routing takes long (even if only in the worst case), then this has certain influences on our timing model. Clearly, if we allow a server to work on a task for, say, 1 second during a step, and routing takes up to, say, 5 seconds, then this would imply undesirable idle time while a server waits for the communication to finish. Therefore, we would like to allow the servers to work on tasks at least as long as all the overhead balancing work takes in the worst case. Only then we have an efficient balancing algorithm.

Our algorithms are local in nature, that is, there is no global instance making certain decisions. All servers decide on their own what to do, although they do not do this without any information regarding the overall load situation in the system. As we will see later, servers can query (few) other servers for information regarding their load, and then, based on this information, estimate the overall system load. In turn, based on this estimation, they decide whether they ought to give away some of their tasks, or are able to accept additional tasks from other servers.

The load balancing problems we are interested in are dynamic, that is, new tasks are generated and old ones are serviced as time passes. We are interested in the long term behaviour of our algorithms, which means that we show that no matter when we observe the system, the critical parameters (which are given below) are "okay" (this is formalised in Section 1.6 below).

In our model the tasks come with a preselected location, due to the fact that they are generated on the servers themselves. Together with the fact that the servers are the (only) ones that also actually service the tasks, this implies that there is no principal necessity to move tasks around – but if there were no load balancing, depending on the load generation scheme, this could lead to highly undesirable load distributions within the system.

As mentioned before, there are several interesting measures that are used to evaluate the performance of load balancing strategies. We are interested in minimising the maximum load of any server (under the load of a server we generally understand the number of tasks in its queue). This can but does not necessarily imply maximisation of the throughput, or minimisation of overall idle time, for instance. Anyway, in this work we do not consider other measures besides the one stated above, except that we are (of course) interested in keeping the overhead due to balancing work as small as possible — what good is a balancing strategy that paralyses the machines? Especially if we transfer tasks due to balancing actions, we do not "spread them wildly" but transfer them in batches and do so only if it is necessary (our algorithms are threshold based, that is, servers transfer tasks if and only if their load exceeds some certain bound). This is the major difference to approaches modelled with balls into bins games (or any approach where tasks come without initial placement), where any newly generated task has to be transferred in any case.

We assume the tasks to be independent. More precisely, in general we do not care about whether they are dependent or not. Our algorithm does its best to keep tasks generated on one server together as long as possible, though. This proves helpful if there are in fact dependencies such that, for instance, tasks need to communicate either while running or if some task needs results of some other task after this one has finished its service. But, as stated before, we do not explicitly take care of that.

All our algorithms are randomised. This basically involves the assignment of balancing partners and the selection of servers to query their load in order to obtain an estimation of the system load. Our so-called adversarial generation model is truly adversarial in that it would force any deterministic load balancing algorithm conforming with our model of computation to produce unacceptable results as far as our parameters are concerned, as we will see later.

SECTION 1.3

Previous Work

1.3.1 Balls Into Bins Approaches

Load balancing problems are frequently modelled by so-called "balls into bins games". These can be either static or dynamic games. Furthermore, one distinguishes sequential and parallel games. In the case of sequential games, the balls (the analogue to tasks) one after the other select $d \geq 1$ bins each (the analogue to machines) at random and then are placed into one of them. In the parallel case, the placement of the balls into the bins happens simultaneously. Usually, the subject of the game is to minimise the maximum load (number of balls) in any bin. Note that balls into bins games are typical members of the "tasks come without a preselected location" class. The reader not familiar with this kind of process might wonder why the balls are not placed using a simple round-robin approach: the first ball into the first bin, the second ball into the second bin, and so on, for this would result in the best possible distribution of balls in bins. But the usual assumptions are that the balls allocate themselves without any coordinating instance, they are indistinguishable, that is, the i-th ball simply doesn't know that it is the i-th ball, and they do not communicate among each other (note that they may very well communicate with the bins they chose at random; more details are given below). Clearly, under these assumptions the round-robin approach does not work.

Why do we present known results concerning balls into bins games, where our approach looks fundamentally different? The answer simply is that the balls into bins games can be considered to be the most simple (randomised) algorithmic approach imaginable and we would like to compare some parameters to the ones of our approach. Clearly, one could implement an algorithm similar to ours using a balls into bins game: play a parallel balls into bins game with all the balls generated during one step (or one phase), and then another one for the next step (phase), and so on.

Basically, there are three methods to analyse such games nowadays. There is the method of layered induction (see [ABKU94]), the witness tree argument (see [MSS95]), and the asymptotic differential equation method

(see [VDK96, Mit96b]). We won't go into detail here, but present a few important results known so far.

It is folklore and quite easily proven that in the case where $m = n$ balls are thrown randomly (and sequentially) into n bins with $d = 1$ (that is, a ball tosses just one die), there will be one bin getting $\Theta(\log n / \log \log n)$ balls with a probability of $1 - o(1)$. For some time this was the best known method of allocation subject to the restrictions as described above.

Karp, Luby, and Meyer auf der Heide ([KLM92]) were the first to present a process using several possible locations (read: $d > 1$) in order to lower the maximum load in the context of simulating parallel random access machines (PRAMs) on distributed memory machines (DMMs). Utilising two randomly chosen hash functions, they obtain a parallel allocation process that guarantees a maximum load of $\mathcal{O}(\log \log n)$ when distributing n memory accesses among n memory modules (equivalent to allocating n balls into n bins). Note that by simply increasing the number of random choices by *one* an exponential drop in the maximum load could be achieved.

Azar, Broder, Karlin, and Upfal ([ABKU94]) examine a similar protocol in a sequential setting. Each of n balls is allowed to choose $d \geq 2$ bins and is placed into the bin with the lowest load among the chosen bins. They show that w.h.p. the maximum load decreases exponentially to $\Theta(\log \log n / \log d)$. Furthermore, they provide results for the dynamic version of their sequential process.

According to the approach of choosing several locations for each ball, much work has been done in analysing balls into bins games in many different ways. For sequential games see [Mit96a], [Mit97] [CS97], [Czu98], [RS98], [CFM+98]. For parallel games see [ACMR95], [Ste96a], [ABS98], and [BMS97]. In the last work, the authors present the first analysis of a parallel balls into bins game for weighted balls, that is, the balls no longer all have the same weight but can have different weights (corresponding to, for instance, varying running times of tasks).

Very recently, there have been two more interesting works on sequential allocation. In [Vöc00], Vöcking decreases the maximum load even further; he shows a bound of $\mathcal{O}(\log \log n / d)$ using a non-uniform way to select the locations of the n balls which creates an asymmetric assignment of balls to bins. In [BCSV00], Berenbrink, Czumaj, Steger, and Vöcking investigate

the case where $m \gg n$. They show a bound for the maximum load of any bin of $m/n + \log\log n + \mathcal{O}(1)$, thus the excess compared to the average load is not dependent on m, the number of balls.

In [Mit96b, Mit96c], Mitzenmacher analysed a dynamic allocation strategy using a differential equations method. Tasks arrive as a Poisson stream at a set of n servers. Each task chooses $d = 2$ servers at random and joins the least loaded among them. He shows that for any interval of fixed length T the expected waiting time of any task is $\mathcal{O}(1)$ (for $n \to \infty$), and that the maximum queue length is $\log\log n/\log d + \mathcal{O}(1)$, with high probability. In [Mit97], he extends his results to varying load generation and service distributions.

Independently, in [VDK96], Vvedenskaya, Dobrushin, and Karpelevich use a similar method to analyse similar systems. This work was extended in [VS97] by Vvedenskaya and Suhov.

In [BCFV99], Berenbrink, Czumaj, Friedetzky, and Vvedenskaya analyse the behaviour of a dynamic parallel allocation process by reducing it to a sequential one, which then is analysed my means of differential equations. The equivalence (or similarity) of the parallel and the sequential process have been shown by simulations only, however.

1.3.2 Other Approaches

Now we present some results obtained in the area of "pure load balancing" which do not employ typical balls into bins strategies.

In [KZ88], Karp and Zhang present a randomised approach to parallelise sequential branch-and-bound algorithms. They show that the execution time of the algorithm expectedly does not exceed an inherent lower bound by more than a constant.

In [RSU91], Rudolph, Slivkin-Allalouf, and Upfal present a rather simple strategy that equalises the load of two processors in one step. They assume a load generation model where at each time step the load variance of any processor due to local generation and service is bounded by some constant. The algorithm works such that each time a server accesses its queue, it decides to initiate a balancing action with a probability inversely proportional to the size of the queue. It then randomly selects some other

processor and these two then equalise their load. The authors show that the expected load of any processor at any point of time is within a constant factor of the average load.

In [LM93], Lüling and Monien use a similar load generation model. A processor initiates a load balancing action if its load has doubled since its last balancing action. To balance load, such a processor chooses a constant number of other processors at random, whereupon they all equalise their load among each other. Lüling et al. show that the expected load difference of any two processors is bounded by a constant factor. Later, Lüling shows in [Lül96] places a bound on the variance on the distribution of the load of the servers.

In [Lau95], Lauer presents a load balancing algorithm assuming the average load av of the system is known. He shows that, with a probability that depends on av, no processor has load exceeding $\mathcal{O}(av)$. Note that he obtains a high probability result only for the case $av = \Omega(\log n)$. Additionally, he presents techniques to estimate the average load of the system and extends his results to this case.

In [BL94], Blumofe and Leiserson present a scheduling approach for so-called strict multi-threaded computations, where a computation consists of a set of threads, and each thread can be seen as a sequential ordering of tasks. The authors present an algorithm and show that the expected time to execute such a computation can be bounded by $\mathcal{O}(T_1/P + T_\infty)$, where T_1 is the minimum sequential execution time, and T_∞ is the minimum execution time given an infinite number of processors.

Another field of massive interest in its own right is that of local iterative load balancing, where servers may exchange load with their direct neighbours in the network only. In [Cyb89] and [Boi90], Cybenko and Boillat, respectively, investigate the so-called diffusion based load balancing, a static approach, where at a step a server may exchange load with all of its neighbours. This is iterated until the load is completely balanced. Unfortunately, this leads to a rather slow convergence.

Ghosh, Muthukrishnan, and Schultz ([GMS96]) improve upon the known results by introducing the concept of over-relaxation, where the amount of load sent over a link not only depends on the current load difference but also on the history of the load transfer over this link so far.

In [GLM+99], Ghosh, Leighton, Maggs, and Muthukrishnan present an algorithm where at each step a server sends one task to each of its neighbours having at least a certain number fewer tasks. They analyse the algorithm in terms of the time needed to decrease the maximum load difference to an arbitrary value, depending on the size, degree, and edge expansion of the network. In [MR98], Muthukrishnan and Rajaraman extend the results to the case of an adversarial dynamic load generation.

1.3.3 Practical Work

Several algorithms and complete tools for dynamic load balancing have been developed and properly examined with the help of experimental studies, for example see [HS97], [WHV95], [DHB97], and [MD96]. For an overview consult [SS97].

SECTION 1.4

General Survey of our Algorithms

In this section we present an overview over the basic structures of our algorithms. We assume that time consists of discrete steps. We further introduce the notion of *phases*, where a phase is nothing but T consecutive steps (T will depend on the generation model and the algorithm in question). A phase, in turn, is divided into four sub-phases as follows.

Load estimation. In this sub-phase the servers compute an estimation of the complete system load. This is done by querying relatively few other servers for their load and then just calculating the average. Note that this sub-phase only applies to the adversarial model and the algorithm devised there. There is no necessity for a server to estimate the system in the stochastical models.

Classification. In this sub-phase the servers classify themselves as light or heavy (or neutral, if nothing else applies). Depending on which generation model we assume, this classification is based either on their current load alone, or on the system load estimation obtained during the previous

load estimation sub-phase. Servers having classified themselves as heavy are supposed to benefit from passing away some of their tasks, whereas light servers are supposed to be able to accept some additional tasks.

Assignment. Here, the heavy servers basically try to find light ones such that any light server is assigned to at most one heavy server, and every heavy server gets assigned exactly one light server. The approach involves some tree-like search mechanism in that a heavy server asks a constant number of other servers whether they are able to accept additional tasks (in other words, whether they are light). This we call a *request*. If there is no light server among the asked ones, in a next step they in turn ask a certain constant number of other servers each in order to "assist" the heavy server in its search for a balancing partner, that is, they also issue requests. This is repeated until every heavy server has been exclusively assigned a light one (with high probability).

Transfer. In this sub-phase we just transfer tasks according to the assignment found above. The number of tasks transferred can be fixed or depend on the heavy server's load, depending on the generation model.

SECTION 1.5

Load Generation and Service Models

We introduce two basic load generation/service schemes and devise load balancing algorithms for both of them. The two schemes are very different in nature in that in the first one the generation and service of tasks obeys some stochastical distribution, whereas in the second one we assume an adversarial model (probably the term "almost arbitrary" is somewhat closer to the true nature, since the word "adversary" seems to have some special reserved meaning throughout the computer science literature — anyway).

1.5.1 The Stochastical Model

One of our main generation models is stochastical in nature in that on a server new tasks are generated and already present ones are serviced based

on random distributions. Common to all our stochastical models is that, in a plain non-balancing system, they yield an expected total system load of $\Theta(n)$ at any arbitrary but fixed point of time. Note that this property implies the stability of the system.

Our main stochastical model allows a server to generate and service one task per step with a certain probability each. In order to meet stability criteria, we have to allow for consumption of a task with at least a slightly larger probability than for generation. Otherwise, as is well known in queueing theory, the number of tasks could not be bounded. We now introduce our first generation model, StochBernoulli(p, ϵ).

StochBernoulli(p, ϵ)

At every time step each server generates one task with a probability of p and consumes one task with a probability of $p(1 + \epsilon)$ (provided there is at least one task) for an arbitrary constant $\epsilon > 0$ such that $p(1 + \epsilon) \leq 1$

Note that given the StochBernoulli(p, ϵ) generation model the service time of a task may very well be longer than just one step, if the server in question throws more than one "don't consume" dice in a row. We model a single server as a FIFO (*first in first out*) queue with attached service station, so whenever the server keeps at least one task, we have that one task is in service in any case. The coin tosses for "service" just determine the service time of the task currently in the service station. Now whenever a "service" coin is flipped with a positive result, the task currently serviced leaves the service station at the end of the step and a new one (if any) immediately moves from the queue into the service station.

We can derive some generation schemes related to StochBernoulli(p, ϵ), and the analysis of the algorithms has to be modified only slightly in order to achieve (almost) the same results. For instance, for the following two generation models we can show results similar to the one for generation model StochBernoulli(p, ϵ). The basic difference between them and model StochBernoulli(p, ϵ) is that now

1. we no longer have that the running time of a task might be longer than just one step, but that now the service time of any task is one

(due to the fact that servers deterministically service a task as long as at least one is present), and

2. two or more (up to some constant number) tasks might be generated during a single step.

Here now follows the description of the variants of our basic generation model StochBernoulli(p, ϵ).

StochGeometric(Δ)

Let $\Delta \geq 1$ be an integer constant. Then, in each time step, every server is allowed to generate up to Δ tasks, obeying the following distribution. For $i \in \{1, \dots, \Delta\}$ a server generates i tasks with a probability of $(1/2)^{i+1}$, that is, with a probability of $1/4$ it generates one task, with a probability of $1/8$ two tasks, and so on. With the remaining probability ($> 1/2$), no task is generated. Furthermore, each server deterministically services one task if at least one is present.

StochBinomial(Δ, p)

Let $\Delta \geq 1$ be an integer constant, let $p = p(\Delta)$ such that $\Delta \cdot p < 1$. For $0 \leq i \leq \Delta$, in each time step a server generates i tasks with a probability p_i of

$$p_i = \binom{\Delta}{i} \cdot p^i \cdot (1-p)^{\Delta - i},$$

as long as $\Delta \cdot p < 1$ ($\Delta \cdot p$ is the expected number of tasks generated per step and server). Each server deterministically services one task per step, if at least one is present.

As mentioned before, StochGeometric(Δ) and StochBinomial(Δ, p) both imply constant service time but more than one task can be generated per time step.

1.5.2 The Adversarial Model

In this section we turn our attention from stochastical load generation models to an adversarial one. The basic idea here is that we assume some "adversary" who is allowed to change the load of any server by up to some

constant Δ into any direction, that is, everything from $+\Delta$ to $-\Delta$ is allowed. Note that this does not exclude, for instance, the generation of $100 \cdot \Delta$ tasks and the servicing of $99 \cdot \Delta$ tasks, respectively. We just need to place a bound on the relative difference between the numbers of tasks generated and serviced. Insignificant for the analysis, we as well may allow the adversary to change a server's load by at most $\Delta \cdot T$ during any interval consisting of $T = (\log \log n)^2$ steps.

These models allow for highly "asymmetric" load generation and servicing schemes. Assuming stochastical schemes we always could make use of the fact that even in a non-balancing system the load distribution among the servers tends to be more or less well-behaved (although, as we will see later, even there without balancing actions the outcome is not precisely desirable). But now the adversary might well decide to have just one server generate tasks during each time step and never consume any, and the rest do nothing but service tasks as long as they have any. This could be used to model the well-known *farmer–worker* approach or variants thereof. Clearly, assuming such a generation scheme, no task would ever be serviced in a non-balancing system, which would have the obvious influences on the evolution of

- the *system load*, which would tend to infinity over time,
- as well as the *ratio of maximum load to average load*, which would tend to n over time (see Section 6.6 for details).

Therefore, it is obvious that in such a case a load balancing algorithm is needed which distributes the tasks generated on the "farmer" server among the "worker" servers.

As mentioned before, we introduce two different kinds of adversarial generation schemes (but this distinction does not make any difference as far as the analysis of the algorithm is concerned, as we will see later). The latter scheme is included here in order to just be somewhat closer to how adversaries are usually defined in the literature, whereas the former one, a step-based one, is closer to how we have defined our stochastical generation schemes.

ADVSTEP(Δ)
: Let $\Delta \geq 1$ be an integer constant. At each time step each server can change its load due to load generation and servicing by up to Δ tasks.

ADVINTERVAL(Δ)
: Let $\Delta \geq 1$ be an integer constant. During any interval of $T = (\log\log n)^2$ consecutive steps, a server is allowed to change its load due to load generation and servicing by up to $\Delta \cdot T$ tasks.

SECTION 1.6

New Results

In this section we introduce the results presented in this thesis, as usual split into stochastical and adversarial generation schemes, respectively.

1.6.1 The Stochastical Model

In this section we present new results concerning our stochastical load generation schemes. Our main model here is STOCHBERNOULLI(p, ϵ), but we present results for STOCHGEOMETRIC(Δ) and STOCHBINOMIAL(Δ, p) as well. The theorems presented correspond to our two different balancing algorithms for the stochastical generation, namely ALGSTOCHMULTICOLL and ALGSTOCHSINGLECOLL; see Chapters 3 and 4, respectively. For definitions of the generation schemes, refer to Section 1.5.

Algorithm ALGSTOCHMULTICOLL

This section deals with our first algorithm for the stochastical generation model, algorithm ALGSTOCHMULTICOLL. The following Theorems 1.1, 1.2, and 1.3 are due to Berenbrink, Friedetzky, and Mayr [BFM98].

Theorem 1.1 (Maximum Load I)
Let $\alpha \geq 1$ be an arbitrary constant, let $0 < p < 1$ and $\epsilon > 0$ such that $p(1+\epsilon) \leq 1$. Assume a load generation probability of p and a load servicing probability of $p(1+\epsilon)$.

Given load generation model StochBernoulli(p, ϵ), *algorithm* AlgStochMultiColl *ensures that with a probability of at least $1-1/n^{\alpha}$ the maximum load of any server is bounded by $(\log\log n)^2$ at any arbitrary but fixed point of time.*

The next theorem states the corresponding results for the load generation schemes StochBinomial(Δ, p) and StochGeometric(Δ).

Theorem 1.2 (Maximum Load II)
Let $\alpha \geq 1$ and $\Delta \geq 1$ be arbitrary constants, let $p \in (0, 1)$ such that $\Delta \cdot p < 1$.

Given generation model StochGeometric(Δ) *or* StochBinomial(Δ, p), *algorithm* AlgStochMultiColl *ensures that with a probability of at least $1-1/n^{\alpha}$ the maximum load of any server is bounded by $\Delta \cdot (\log\log n)^2$ at any arbitrary but fixed point of time.*

We additionally show that given balancing algorithm AlgStochMultiColl, the expected number of balancing requests as introduced in Section 1.4 (and expanded on in Section 3.1) for any processor trying to find a balancing partner is constant for all three generation models StochBernoulli(p, ϵ), StochGeometric(Δ), and StochBinomial(Δ, p).

Such a request involves communication to at most some constant number of other servers, thus the communication due to the establishment of an assignment from a heavy to a light processor (one which needs to give away tasks and one which is able to accept additional tasks, respectively) expectedly also is constant.

Theorem 1.3 (Balancing Requests I)
Let $\alpha \geq 1$ be an arbitrary constant.

Given either the StochBernoulli(p, ϵ), *the* StochGeometric(Δ), *or the* StochBinomial(Δ, p) *generation model, algorithm* AlgStochMultiColl *ensures that the expected number of balancing requests needed to find a balancing partner is constant for each heavy server.*

Algorithm AlgStochSingleColl

The second algorithm for the stochastical generation model, AlgStochSingleColl, works somewhat different from algorithm AlgStochMultiColl. Although technically more complicated, it allows us to decrease the length of any phase from $\Theta((\log\log n)^2)$ to $\Theta(\log\log n)$, which, in turn, also reduces the bound on the maximum load of any server by the same factor. The following Theorems 1.4 and 1.5 are due to Berenbrink, Friedetzky, and Steger [BFS99].

Theorem 1.4 (Maximum Load III)
Let $\alpha \geq 1$ be an arbitrary constant, let $0 < p < 1$ and $\epsilon > 0$ such that $p(1+\epsilon) \leq 1$. Assume a load generation probability of p and a load servicing probability of $p(1+\epsilon)$.

Given load generation model StochBernoulli(p, ϵ), *algorithm* AlgStochSingleColl *ensures that with a probability of at least $1 - 1/n^\alpha$ the maximum load of any server is bounded by $\mathcal{O}(\log\log n)$ at any arbitrary but fixed point of time, where the constant depends on α, p, and ϵ.*

Again, we can show similar results for the stochastical generation models StochGeometric(Δ) and StochBinomial(Δ, p).

Theorem 1.5 (Maximum Load IV)
Let $\alpha \geq 1$ and $\Delta \geq 1$ be arbitrary constants, let $p \in (0,1)$ such that $\Delta \cdot p < 1$.

Given generation model StochGeometric(Δ) *or* StochBinomial(Δ, p), *algorithm* AlgStochSingleColl *ensures that with a probability of at least $1-1/n^\alpha$ the maximum load of any server is bounded by $\mathcal{O}(\log\log n)$ at any arbitrary but fixed point of time, where the constant depends on α and Δ (and p in the case of* StochBinomial(Δ, p)*).*

Finally, we briefly state some recovery property of the systems we investigate. This recovery property is inherent to the load generation models, and consequently influences of the algorithms can be neglected entirely, which implies that the statement is valid for both of our algorithms AlgStochMultiColl and AlgStochSingleColl dealing with the stochastical load generation. What we mean with recovery is that if the system is

in some arbitrary state at some point of time (arbitrary here means that the $\mathcal{O}(n)$ bound on the system load we require for the analysis does not hold), then the system will "automatically" tend to some typical state, where said bound will be valid again.

Comparison to Balls into Bins Games

To our best knowledge, Theorem 1.4 is the first result that provides the same asymptotical bounds holding with high probability for dynamic load balancing algorithms as have been known for dynamic balls into bins games of a certain class. Of course one cannot compare apples and oranges, so the balls into bins games we compare our algorithm to have to somehow correspond to our load generation and consumption behaviour, respectively, in that

- in our approach, there there $\mathcal{O}(n)$ tasks generated per step, and $\Omega(n)$ tasks serviced; resulting in an upper bound on the maximum load of any server of $\mathcal{O}(\log\log n)$, whereas
- for balls into bins games, $\mathcal{O}(n)$ balls are thrown into n bins with $d = 2$ choices for each ball per step, and also $\Omega(n)$ balls are deleted; resulting in a maximum load of $\mathcal{O}(\log\log n)$ (see[ABS98]).

Previous results either provided only expected bounds (or analysed the variance of certain parameters, at best), or needed the system load to be sufficiently large, where in our case a system load of $\mathcal{O}(n)$ is sufficient.

We can additionally compare some other parameters of our algorithms to parallel dynamic balls into bins approaches. There, every ball inevitably upon generation (more precisely, after some communication) is thrown into some randomly chosen bin. This implies $\Omega(n)$ communication messages for the n balls generated per step, and $\Omega(1)$ messages per ball.

Our algorithms are threshold based and transfer tasks in "packets", which reduces the communication complexity significantly. Our algorithm ALG-STOCHMULTICOLL requires $\mathcal{O}(n/\log n)$ messages in order to "allocate" the $\mathcal{O}\left(n(\log\log n)^2\right)$ tasks generated in a phase of length $\Theta((\log\log n)^2)$. This implies also $\mathcal{O}(n)$ tasks generated per step but just $\mathcal{O}(1/\log n)$ messages per task on average, yielding an improvement of a factor of $\Omega(\log n)$.

1.6.2 The Adversarial Model

So far, when assuming a stochastical generation model, we could exploit an implicit upper bound of $\mathcal{O}(n)$ on the system load both in the non-balancing system as well as in the balancing one, w.h.p. (see Lemmas 3.1, 3.3, and 3.4, respectively, for details). Of course, this no longer holds for the adversarial load generation (ADVSTEP(Δ) and ADVINTERVAL(Δ), respectively), since here the system load can grow to infinity as time passes (recall that in each time step, a server may increase its load by up to some constant Δ). Therefore, in order to enable the algorithm to make sensible decisions concerning the classification of the servers as heavy, light, or neutral, we need to employ the load estimation sub-phase as briefly described in Section 1.4.

In order to be able to present the new results for the adversarial model, we need to introduce some definitions. We divide time into subsequent phases of length $T = \Theta((\log\log n)^2)$. Let τ be the first time step of some fixed phase, and let M_τ be the system load at this time step τ. Let $\lambda > 0$ be some constant. The system is said to be λ-*balanced* for this phase if there is no processor with load exceeding $\lambda \cdot (M_\tau/n + 12\Delta(\log\log n)^2)$ at time step τ. Since the net load gain of any processor during one phase is bounded by ΔT, then if the system is λ-balanced, the load of the processors is at most a constant factor from the average at every step of the phase.

Again, the balancing algorithm ALGADV for the adversarial generation model follows the algorithm sketch from Section 1.4. The following Theorems 1.6 and 1.7 which present results concerning the performance of algorithm ALGADV are due to Berenbrink, Friedetzky, and Steger [BFS99].

Theorem 1.6 (Maximum Load V)
Let $\Delta \geq 1$, $\alpha \geq 1$, $\beta \geq 1$, and $\ell > 0$ constants, such that $\alpha = \beta + \ell$. Let $\lambda = 20 \cdot (8\alpha + 16)^{2\alpha+5}$.

If the system is λ-balanced for the i-th phase, then, given either generation model ADVSTEP(Δ) or ADVINTERVAL(Δ), with a probability of at least $1 - 1/n^\alpha$, algorithm ALGADV ensures that it is λ-balanced for the $(i+1)$-th phase, and that with a probability of at least $1 - 1/n^\beta$ it remains λ-balanced for the next n^ℓ phases.

Again, we can bound the expected number of balancing requests issued per heavy processor. This is stated in Theorem 1.7.

Theorem 1.7 (Balancing Requests II)
Let $\Delta \geq 1$ be an arbitrary constant.

Given either the AdvStep(Δ) *or* AdvInterval(Δ) *generation model, algorithm* AlgAdv *ensures that the expected number of balancing requests needed to find a balancing partner is constant for each heavy server.*

CHAPTER 2

The Collision Protocol

The collision protocol (also often called “collision game”) is the basic building block of all of our algorithms. It has its roots in the context of shared memory simulations, but it is applicable in a wide range of problems. Its basic idea is to distribute a number of randomly chosen access requests such that in the end there is an almost even distribution of the requests among their destinations. It can, for instance, not only be utilised to distribute accesses to copies of shared memory cells which are stored in the modules of a distributed memory machine (DMM) among these modules, but also for allocating balls into bins (in this case the balls are the accesses, and the bins are the memory modules).

The basic idea of the collision game is rather simple and elegant. We now briefly describe it in a more or less abstract context (but still close to how we are going to make use of it). Consider a system of n *sources* and n *destinations*, where each source requests to access a certain number b of destinations, but no destination wants to be accessed by more than c sources. In the context of shared memory simulations, the sources are the processors of the DMM, and the destinations are the memory modules; whereas in the context of balls into bins games the sources are the balls and the destinations are the bins.

The (a, b, c)-collision game now distributes the requests of the sources among the destinations such that the aforementioned restrictions are fulfilled. Ever source randomly chooses $a > b$ destinations and sends so-called *queries* to each of them (the set of queries sent from one source we call

request). Now, the protocol proceeds in rounds as described in Figure 2.1. Obviously, *if* the protocol terminates, no destination accepts more than c queries (but note that the protocol does not necessarily terminate).

In our algorithms we use the collision protocol in two slightly different settings, depending on c, the collision parameter. In one setting we have to restrict c to one, which clearly means that we cannot allow all the sources to issue requests – even in the case $b = 1$ we would then have to construct a permutation among sources and destinations (if $b > 1$ in this case, an assignment would be impossible at all). Therefore, we slightly modify the protocol, resulting in the (n, ϵ, a, b, c)-collision protocol. The only real differences are that we now have n destinations and ϵn requests, for some $\epsilon \leq 1$, and that we terminate the protocol after a fixed number of rounds. This modified protocol can be found in Figure 2.2. Note that this protocol terminates in any case, but it can happen that not all the requests get fulfilled.

Note that as long as a, b, and c are constant parameters, each source communicates with at most a constant number of destinations in any round. Therefore, if we now identify both sources and destinations with the servers we are actually concerned with, the servers communicate with at most a constant number of other servers, thus fulfilling the restriction we have placed in Section 1.2.

We now provide two theorems connecting the parameters ϵ, a, b, and c to the number of rounds, r.

Lemma 2.1 (Collision Protocol for $c > 1$)
Let $\alpha \geq 1$ an arbitrary constant, let $a \geq 2$, $b = a - 1$, $\epsilon = 1$, and $c \geq \max\{8ea^2, 4(\alpha + 2)\}$.
With a probability of at least $1 - 1/n^\alpha$, the (n, ϵ, a, b, c)-collision protocol terminates with a valid assignment of queries after $r \geq \frac{\log\log n}{\log c} + 1$ rounds.

Lemma 2.2 (Collision Protocol for $c = 1$)
Let $\alpha \geq 1$ an arbitrary constant, let $c = 1$, $b \geq 2$, $a = b + 2\alpha + 2$, and $\epsilon \leq 2(2a)^{-(2\alpha+3)}$.
With a probability of at least $1 - 1/n^\alpha$, the (n, ϵ, a, b, c)-collision protocol terminates with a valid assignment of queries after $r \geq \frac{\log\log n}{\log 2\alpha+2} + 2$ rounds.

The proofs for both theorems can be found in [Ber00], similar proofs in, e.g., [MSS95] or [Ste96b]. Actually, in [MSS95] the authors present an analysis of an even more general version of the collision protocol; they obtain a probability bound of $1 - 1/n^{\beta}$ for some $\beta < 1$, which unfortunately is too weak for our purposes.

Now, we just briefly outline the structure of the proofs. Basically, they are build around a so-called *witness tree* argument. Suppose that there is at least one unfulfilled request left after the termination of the protocol (after r rounds). This implies that less than b of its a queries have been accepted during the r rounds of the protocol, which, in turn, implies that there is at least one destination (with one of the queries in question directed to it) with at least c queries belonging to other requests directed to it in round r, thus being unable to accept them (recall that queries are accepted if there are at most c of them). Now we can ask the question *why* these queries are still active at all before round r. That is because the requests they belong to have not been fulfilled so far, meaning that for each of them during round $r - 1$ at least one query was directed to a destination unable to accept. This argument can be continued until we come to round 0 (to the very first issuing of queries). This way we construct a witness tree where the nodes are destinations unable to accept and the edges are at queries issued to these destinations. The tree has depth r and a degree of at least $c + 1$.

Strongly simplifying the matter, the technically involved proof now basically shows that the occurrence of such a witness tree is highly unlikely, implying that it is unlikely that there is a still non-fulfilled request after r rounds of the protocol. As mentioned before, see [Ber00] for details.

- Each source ("each request") randomly chooses $a > b$ pairwise distinct destinations and sends queries to them (which are called *active*).
- While there are active queries:
  - Each destination with at most c queries directed to it accepts them. It then leaves the game and never again accepts any further query.
  - Each source which so far (accumulatively) has at least b of its a queries accepted also leaves the game and cancels all remaining queries.
  - The remaining queries are re-sent *without making new random choices.*

Figure 2.1: The (a, b, c)-collision protocol

- For each of the ϵn access requests randomly choose $a > b$ pairwise distinct destinations and send queries to them (which are called *active*).
- For r rounds do:
  - Each destination with at most c queries directed to it accepts them. It then leaves the game and never again accepts any further query.
  - Each request which so far (accumulatively) has at least b of its a queries fulfilled also leaves the game and cancels all remaining queries.
  - The remaining access queries are re-sent *without making new random choices.*

Figure 2.2: The (n, ϵ, a, b, c)-collision protocol

CHAPTER 3

The Balancing Algorithm ALGSTOCHMULTICOLL

In this chapter we discuss our first balancing algorithm for the stochastical load generation model, algorithm ALGSTOCHMULTICOLL. As is stated in Theorem 1.1 on page 18, it obtains a maximum load of $(\log\log n)^2$ for generation model STOCHBERNOULLI(p, ϵ), w.h.p. Following the algorithm sketch from Section 1.4, time is divided into subsequent phases. Here, a phase consists of $T = 1/10 \cdot (\log\log n)^2$ steps. Every phase is split into the four sub-phases load estimation, classification, assignment, and transfer.

We have mentioned before that the load estimation sub-phase is not always needed. Actually, this is the case for all of our algorithms for stochastical load generation schemes, since here we can place a bound of $\mathcal{O}(n)$ on the total system load, w.h.p. Therefore, instead of doing the classification as heavy, neutral, or light with respect to some estimation of the system load, servers can do so based on some fixed value that does not change over time. The intuitive reason is that if the system load does not change too much over time, there is no need to reflect such a change in the decision as far as classification is concerned.

SECTION 3.1

The Algorithm

In this section we introduce our algorithm AlgStochMultiColl. Each phase has a length of $T = 1/10 \cdot (\log\log n)^2$. Note that there is no load estimation sub-phase. A single phase now looks as follows.

Classification. Servers having a load of at most T at the beginning of the phase are classified as *light*, and servers with a load of at least $8T$ at the beginning of the phase are classified as *heavy*. The remaining servers are classified as *neutral*. Note that the thresholds are not with respect to some "floating" value but fixed and only dependent on T.

Assignment. The main goal of the assignment sub-phase is to assign one light server to every heavy one, where any light server may be assigned to at most one heavy server. The assignment sub-phase proceeds in $t = \log\log n/\log b$ rounds. Let our description start with the very first round.

We say that every server classified as heavy *initiates* a balancing action (or initiates the search for a balancing partner). We now play an $(n, \epsilon, a, b, c = 1)$-collision game, where only heavy servers issue requests. The idea is to for some heavy server P to try to find a light balancing partner among the b servers that have accepted a query that belongs to P's request (recall that a request consists of a queries from which b will be accepted if the protocol is successful). Note that since $c = 1$, we will have to place a strict bound on the number of concurrent requests, and that also due to $c = 1$, if a light server has accepted a query then this is the *only* query it has accepted, and therefore it will not have to decide which heavy server to "contact".

Now that the collision game is played, every heavy server P has b distinguished servers which have accepted a query belonging to its request. Assume that together with the query the identifier of P was communicated, so servers having accepted a query know which heavy server (that has initiated a balancing action) it belongs to. Now every light server having accepted a query sends an *offer message* to the corresponding heavy server (recall that due to $c = 1$ there can be at most one).

Now every heavy server having received such an offer message arbitrarily selects one of them. For this purpose we again assume the 1-collision rule, that is, independent of the number of messages actually arriving, one of them can be evaluated fast. Now every heavy server having received an offer message sends back an *acknowledgement message* to the selected light server. From now on those two are defined to be balancing partner, and during the next rounds of this sub-phase these light servers are considered to be *non-available*.

Of course it can happen that there are heavy servers which have not found a light partner during this first round of this sub-phase. Instead of playing the same game again, we now let the servers which have accepted a query but have not been light issue requests for another collision game. The intuition behind this is that we spread the search for every so far not successful heavy server from round to round, thus building some kind of *query tree*. A heavy server P initially issues a request. If none of the servers having accepted is light, each of them issues a request on behalf of P during a next collision game. At this point of time there are not b but now b^2 servers among a light one can be searched for. If again there is no light server among those, each of those b^2 servers issues a request on behalf of P during a next collision game, and so on.

There are just some technical details to take care of. Each time a light server accepts a query, the mechanism as described above comes into play: it sends an offer message to the heavy server on whose behalf the query was sent. This then arbitrarily selects one of the light servers having sent such a message and in turn answers with an acknowledgement message.

It can now happen that a light server was selected to be balancing partner, and in some later round again accepts a query. This is why assigned light servers are considered to be non-available. Although light, they must not be assigned to some heavy server more than once. So as far as finding a partner is concerned, they are not seen as light any longer.

There is one further thing we can do to reduce the communication significantly. As the protocol is defined so far, each time a server having accepted a query and not having received an acknowledgement message in turn issues another request for the next collision game. Obviously, this is not necessary if the heavy server on whose behalf the query has been sent in

fact has found a balancing partner.

The most efficient solution now is to send a *stop searching* message to all the servers which have accepted a query which was sent on its behalf. This can be done as follows. As soon as a collision game is finished, each heavy server P having initiated a balancing action waits for offer messages. If no such messages arrives, it sends *continue searching* messages "down" to the servers which have accepted its queries during the very first round. These in turn forward the message to the servers having accepted queries on behalf of P during the second round, and so on. This way, after at most $\log\log n/\log b$ hops the message arrives at the servers which are about to participate in the next collision game. Similarly, if the heavy server P has found a partner, it sends down not *continue searching* but *stop searching* messages. As we will see, playing a single collision game takes time $\log\log n/\log b + 2$, so passing the messages does not take longer. A drawback of this solution is that every server has to store the identifiers of the servers which have accepted queries issued by itself. Although due to the $c = 1$ collision rule a server occurs at most once on each level of the trees (actually, it occurs at most once on each level of the complete forest consisting of all the query trees), but it may well occur on each single level of a certain tree. So, to be able to pass down a message from the root of its tree (the server having initiated the balancing action which led to the development of the tree), it would need to store $\mathcal{O}(\log\log n)$ identifiers, resulting in a memory requirement of $\mathcal{O}(\log n \cdot \log\log n)$ bits ($\log n$ bits for each identifier). The same is true if it occurs on each level of the complete forest, no matter in which tree, as long as all the trees involved have to be developed down to the full depth.

For a very small example, refer to Figure 3.1. Here, servers P_i and P_j are classified as heavy and try to find balancing partners. During the first round, P_j is sent offer messages from the light servers P_k and P_m, and P_k is selected to be the balancing partner, whereupon P_k is sent an acknowledgement message. No new requests are issued on behalf of P_j afterward. P_i has not been so lucky and has so far not found a partner, so the search has to go into the second round. Here, a request on behalf of P_i arrives at (light) P_k, but P_k has been selected to become balancing partner already, so the search still has to go on, until in round three P_i finally selects P_l to become its balancing partner.

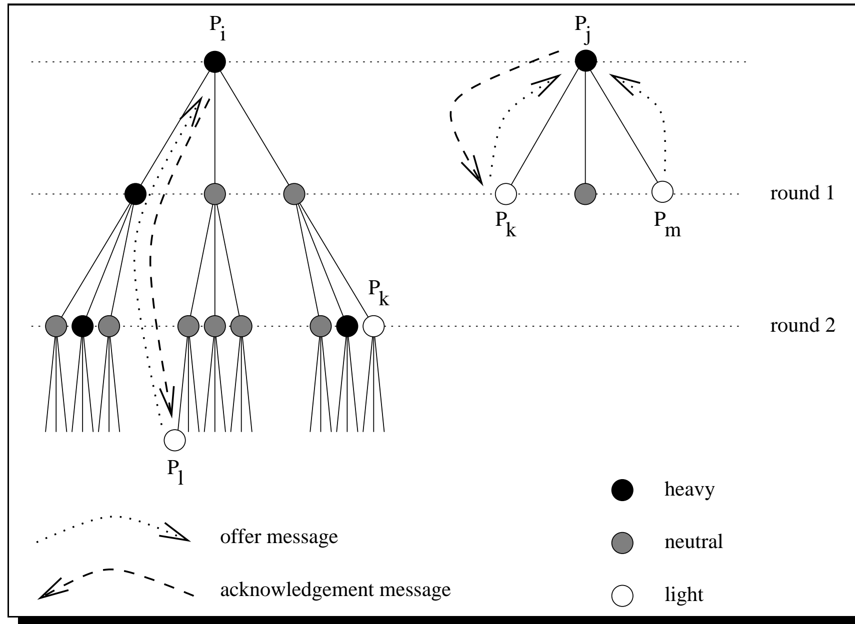

Figure 3.1: Basic idea of AlgStochMultiColl

Transfer. In this sub-phase heavy servers transfer 4T tasks each to their light balancing partners. Tasks to be transferred are both taken from and appended to the tails of the FIFO queues of the servers in question.

SECTION 3.2

The Analysis

In this section we analyse the performance of algorithm AlgStochMultiColl given load generation model StochBernoulli(p, ϵ). The analysis is split into parts as follows.

1. In Section 3.2.1, we analyse the non-balancing system under generation model StochBernoulli(p, ϵ). Here, no balancing actions at all are performed. We will provide an $\mathcal{O}(n)$ bound on the total system load at any point of time in this system (holding with high probability).

2. In Section 3.2.2, we show that this bound also holds for our balancing system, that is, when algorithm ALGSTOCHMULTICOLL actually runs. We need this bound on the system load for the remaining parts of the analysis – furthermore, a load balancing algorithm wouldn't be worth its money if it increased the system load (significantly).

3. Next, in Section 3.2.3, we analyse a single phase of our algorithm. We provide an upper bound on the number of heavy servers as well as a lower bound on the number of light servers. We show that the collision games actually resolve all the requests, and that the assignment from light servers to heavy ones succeeds. All this holds with high probability.

4. In Section 3.2.4, we then prove the main theorem of this part, namely Theorem 1.1 on page 18. This is done basically by concluding that if a server is beyond our bound on its load of $(\log\log n)^2$ then that its first approach of finding a balancing partner must have failed. We then place tight bounds on the probability that such an event could have actually happened.

5. Finally, we consider a few minor aspects. In Section 3.2.5, we show that algorithm ALGSTOCHMULTICOLL performs equally well for generation schemes STOCHGEOMETRIC(Δ) and STOCHBINOMIAL(Δ, p) as it has been shown to do for STOCHBERNOULLI(p, ϵ). Furthermore, in Section 3.2.6, we show that all those generation schemes are "self-repairing" in the sense that even if the system is in some arbitrary state at some point of time it will recover itself and tend to approach a typical state, where we again can conclude the $\mathcal{O}(n)$ bound on the system load – even without (or despite, as skeptics might object. . .) the interaction of balancing algorithm ALGSTOCHMULTICOLL.

3.2.1 The Non-Balancing System

In this section we assume a plain non-balancing system with underlying load generation scheme STOCHBERNOULLI(p, ϵ). We first place bounds on the load of a single server, and then from that conclude statements concerning the complete system load. We always assume that the system starts empty at time step 0.

Lemma 3.1 (Load of a Single Server)
Let $\alpha \geq 1$ and $p \in (0,1)$ be arbitrary constants, let $\epsilon > 0$ such that $p(1+\epsilon) \leq 1$. Let $\mu = \frac{\epsilon+(1-p-\epsilon p))}{1-p-\epsilon p} > 1$.
Given load generation model $\textsc{StochBernoulli}(p, \epsilon)$*, with a probability of at most $(1/\mu)^k$, a server will have a load of at least k at any arbitrary but fixed point of time in a non-balancing system.*

Proof. First, we observe that during a step, a server may either increase its load by one (one task generated and none serviced), decrease its load by one (no task generated and one task serviced – presuming that there was at least one), or have its load unaltered (either both generated and serviced a task, or neither generated nor serviced). Let $p^{(+)}$ denote the probability for a net increase, $p^{(-)}$ denote the probability for a net decrease, and $p^{(=)}$ denote the probability for an unchanging load.

As a server generates as task with a probability of p and services a task (if any) with a probability of $p(1+\epsilon)$, we obtain

$$\begin{aligned} p^{(+)} &= p \cdot (1-(p(1+\epsilon))) \\ p^{(-)} &= p(1+\epsilon) \cdot (1-p) \\ p^{(=)} &= 1-p^{(+)}-p^{(-)} \end{aligned}$$

We just have to take special care of the situation of a server having zero tasks. Clearly, it can either increase its load by one (with probability $p^{(+)}$ as defined above), or does not change its load. There is no negative load, so a decrease by one is impossible. Consequently, we have to modify the probability for an unchanging load from $p^{(=)} = 1 - p^{(+)} - p^{(-)}$ to $p_0^{(=)} = 1 - p^{(+)}$ in this case.

Obviously, the evolution of the load of a single server follows a simple one-dimensional birth-death process. To analyse the load situation of a server, we therefore construct a discrete-parameter Markov chain, where state k corresponds to the situation of the server having a load of exactly k (at the beginning of a certain time step). The chain's transition probability

matrix $\mathcal{M} = (p)_{ij}$, $i, j \geq 0$, looks as follows.

$$p_{ij} = \begin{cases} p^{(+)} & \text{if } j = i+1 \text{ (net increase from } i \text{ to } i+1) \\ p^{(-)} & \text{if } i > 0 \text{ and } j = i-1 \text{ (net decrease from } i \text{ to } i-1) \\ p^{(=)} & \text{if } j = i > 0 \text{ (no change)} \\ p_0^{(=)} & \text{if } j = i = 0 \text{ (no change)} \\ 0 & \text{otherwise} \end{cases}$$

Here, p_{ij} denotes the probability for the chain to reach state j directly from state i (or, in our scenario, for a server to reach load j at the end of a step when having had load i at the beginning of this step). Note the different probabilities for transitions $0 \rightarrow 0$ and $i \rightarrow i$ for $i > 0$, respectively. This is due to the reasoning from above (no transition to the left from state 0). A graphical representation of the chain can be found in Figure 3.2.

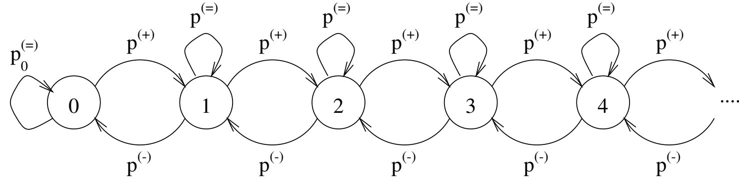

Figure 3.2: The Markov Chain for a single server

It is well known that such a chain has a unique stationary distribution $\vec{v} = (v_0, v_1, \ldots)$ with v_i as the probability for the chain being in state i in the steady state, corresponding to the server having a load of i (see, for instance, [Tri92]). With

$$\frac{p^{(+)}}{p^{(-)}} = \frac{p \cdot (1 - (p(1+\epsilon)))}{(p(1+\epsilon)) \cdot (1-p)} = \frac{(1 - p - \epsilon p)}{\epsilon + (1 - p - \epsilon p)}$$

v_i now can be expressed as

$$\begin{aligned} v_i &= \left(1 - \frac{p^{(+)}}{p^{(-)}}\right)\left(\frac{p^{(+)}}{p^{(-)}}\right)^i \\ &= \left(1 - \frac{(1-p-\epsilon p)}{\epsilon + (1-p-\epsilon p)}\right)\left(\frac{(1-p-\epsilon p)}{\epsilon + (1-p-\epsilon p)}\right)^i \\ &= \left(1 - \frac{1}{\mu}\right)\left(\frac{1}{\mu}\right)^i \end{aligned}$$

with $\mu > 1$ as defined in the lemma. Now let $\mathcal{A}_k$ denote the event that in the steady state the chain is in some state i with $i \geq k$ (corresponding to a server having a load of *at least* k). Clearly,

$$\text{prob}\left(\mathcal{A}_k\right) \leq \sum_{i=k}^{\infty} \nu_i = \sum_{i=k}^{\infty} \left(1 - \frac{1}{\mu}\right)\left(\frac{1}{\mu}\right)^i = \left(\frac{1}{\mu}\right)^k .$$

This finishes the proof of the lemma. □

Now we can prove the main statement of this section, namely that we can place a bound on the complete system load. First, we need to introduce a well-known tail estimate, known as Chernov-Hoeffding-Bounds.

Lemma 3.2 (Chernov-Hoeffding)
Let $n > 0$, let $X_1, \dots, X_n$ be independent random variables with domain $[0, z]$ for some arbitrary $z > 0$. Let $\mu := \mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right]$ be the mean expected value of $X_1, \dots, X_n$. Then, for all $u \geq 1$,

$$\text{prob}\left(\sum_{i=1}^{n} X_i \geq \mu \cdot n \cdot u\right) \leq \left(\frac{e^{u-1}}{u^u}\right)^{\frac{n\mu}{z}}$$

Proofs of Lemma 3.2 can be found in [DM90] or [Hof87], among others. Most notable is that, unlike standard Chernov bounds, here the random variables may have a domain of $[0, z]$ for some arbitrary value $z > 0$ instead of having to be restricted to $\{0, 1\}$ Bernoulli random variables. This is exactly what we need, since if we want to bound the complete system load, we have to express the load of a single server by a random variable.

Unfortunately, we cannot start right away, because the load of a server can become *arbitrarily* large, and Lemma 3.2 requires a fixed bound on the domain of any random variable. Therefore, we need an intermediate step of placing a high-probability bound on the maximum load of any server, which, in our case, is easily done.

Lemma 3.3 (System Load)
Under the conditions of Lemma 3.1, and given load generation model StochBernoulli(p, ϵ), *with a probability of at least $1 - 1/n^{\alpha+1}$, the complete system load (the sum of the load of all servers) is $\mathcal{O}(n)$ at any arbitrary but fixed point of time in the non-balancing system.*

Proof. First note that we have a non-balancing system, so there are no interactions whatsoever between the single servers. Consequently, the random variables describing the load of the servers are completely independent.

Fix an arbitrary time step. Let X_i, $1 \le i \le n$, denote the load of server i at the beginning of this time step. Let $X = \sum_{i=1}^{n} X_i$ denote the complete system load at the beginning of this time step. Clearly, for any $1 \le i \le n$, we have

$$\begin{aligned}
E[X_i] &= \sum_{j=0}^{\infty} j \cdot \text{prob}\,(X_i = j) \\
&\le \sum_{j=0}^{\infty} \text{prob}\,(X_i \ge j) \\
&\le \sum_{j=0}^{\infty} \left(1 - \frac{1}{\mu}\right) \cdot \left(\frac{1}{\mu}\right)^j \\
&= \frac{1/\mu}{(1 - 1/\mu)} \\
&= \frac{1}{\mu - 1}
\end{aligned}$$

Furthermore, by the above and by linearity of expectation, we have

$$E[X] = \sum_{i=1}^{n} E[X_i] \le \frac{n}{\mu - 1}.$$

Since μ is a constant, we can conclude that the expected complete system load is $\mathcal{O}(n)$. Now there is no inherent upper bound on the values of the random variables X_i (the load of a single server *can* become arbitrarily large). In order to apply the Chernov-Hoeffding bounds from Lemma 3.2, we need to artificially cut down the domain of the X_i. For this purpose, let

$$Y_i = \min\left\{\frac{\alpha + 2}{\log \mu} \cdot \log n,\ X_i\right\}$$

for $1 \le i \le n$, that is, Y_i is very much the same as X_i, just that we do not allow it to grow past $\Theta(\log n)$. Furthermore, let

$$Y = \sum_{i=1}^{n} Y_i.$$

By Lemma 3.1, the probability that the i-th server has a load of at least $\frac{\alpha+2}{\log \mu} \cdot \log n$ is at most

$$\left(\frac{1}{\mu}\right)^{\frac{\alpha+2}{\log \mu} \cdot \log n} \leq \frac{1}{n^{\alpha+2}}.$$

Consequently, $X = Y$ with a probability of at least $1 - 1/n^{\alpha+1}$. According to Lemma 3.2, for constants μ and α, we have

$$\operatorname{prob}\left(Y \geq 4 \cdot \frac{n}{\mu - 1}\right) \leq \left(\frac{e^3}{4^4}\right)^{\frac{n}{\mu-1} \cdot \frac{\log \mu}{(\alpha+2) \cdot \log n}} \leq \frac{1}{n^{\alpha+1}}.$$

Since we start with an empty system, this estimation holds for every arbitrary but fixed point of time. □

In this section, we have shown that we can place a bound on the complete system load in a non-balancing system (but note that with a probability of $1 - o(1)$ there *will* be a server with a load of $\Omega(\log n / \log \log n)$).

3.2.2 From Non-Balancing to Balancing Systems

In this section we show that the complete system load in a balancing system also can be bounded, as we have done in the previous section for a plain non-balancing system.

Lemma 3.4 (System load in balancing system)
Let $\alpha \geq 1$ and ℓ_s be arbitrary constants. Fix an arbitrary time step.
With a probability of at least $1 - 1/n^{\alpha+1}$, if the system load in the non/balancing system is at most $\ell_s n$, then this is true for the balancing one also.

Proof. We will define two systems $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{A}$ is a variant of our non-balancing system, and $\mathcal{B}$ is a variant of our balancing system. The only difference between the standard non-balancing system and $\mathcal{A}$ (and between the standard balancing system and $\mathcal{B}$) is that we rearrange the order in which tasks are serviced by a single server. Obviously, this has no influence whatsoever on the load of any server or on the complete system load. Further, assuming generation model StochBernoulli(p, ϵ), we may

decide to preempt a running task. But since we do this only if there is some other task ready to be serviced, and given model STOCHBERNOULLI(p, ϵ), a server decides step by step whether or not a task's service is finished, this also has no influence (a preempted task has the same chance to finish when it is scheduled back into the service station as as when it would never have left).

A last important assumption is that we assume *identified* servers in $\mathcal{A}$ and $\mathcal{B}$, that is, at any time step, the i-th server in $\mathcal{A}$ makes exactly the same random decisions as the i-th server in $\mathcal{B}$.

A server P in system $\mathcal{A}$ has two queues, $P_{\mathcal{A}}^{(1)}$ and $P_{\mathcal{A}}^{(2)}$, and a server P in system $\mathcal{B}$ has three queues, $P_{\mathcal{B}}^{(1)}$, $P_{\mathcal{B}}^{(2)}$, and $P_{\mathcal{B}}^{(3)}$. In both systems, the first two queues always hold self-generated tasks (and whenever a new tasks is generated, it lands in the first queue), whereas in system $\mathcal{B}$, the third queue $P_{\mathcal{B}}^{(3)}$ of any server P holds tasks that P has received due to balancing transfers. In both systems, tasks from the first queue are preferred over tasks of the second queue for servicing (and in system $\mathcal{B}$, tasks of the third queue have lowest priority). This implies that whenever a task coming from a second or third queue is in service, and a new task is generated, then the running task is preempted and the new one starts it service.

Each time a server P of system $\mathcal{B}$ decides to transfer $4T$ tasks (recall the sketch of the algorithm) to some other server Q due to a balancing action, the following happens.

System $\mathcal{B}$. Tasks to be transferred come from the third queue, $P_{\mathcal{B}}^{(3)}$. If there are not $4T$ tasks in this queue, P fills up with tasks first from $P_{\mathcal{B}}^{(2)}$, and if there still are not $4T$ tasks altogether, from $P_{\mathcal{B}}^{(1)}$. Tasks always are inserted into $Q_{\mathcal{B}}^{(3)}$ according to their priority, i.e., first tasks from $P_{\mathcal{B}}^{(1)}$, then $P_{\mathcal{B}}^{(2)}$, and finally $P_{\mathcal{B}}^{(3)}$, and always at the end of the queue. Now, both servers P and Q move all of their first-queue tasks to the front of their second queues (this decouples load balancing from normal generation/consumption, since now the firsts queues are empty, and whenever a new task is generated, it automatically has preference over all existing tasks).

System $\mathcal{A}$. Of course, in system $\mathcal{A}$ there are no load balancing actions at all. But since we assume that servers in systems $\mathcal{A}$ and $\mathcal{B}$ are identified,

whenever some server P in system $\mathcal{B}$ decides to move tasks to some other server Q, then the "doubles" of P and Q in $\mathcal{A}$ move all their tasks from their first queues to their second queues (just like in system $\mathcal{B}$, just without actual load balancing).

By definition of the systems $\mathcal{A}$ and $\mathcal{B}$, the complete system load of our original non-balancing system is just the same as the one of $\mathcal{A}$, and the system load of our original balancing system is just the same as the one of $\mathcal{B}$.

Now fix some task t in system $\mathcal{B}$, which is moved from P to Q due to a load balancing action. Let k denote the number of tasks of P with precedence over t, and let k' denote the number of tasks of P which have precedence, and which are also moved over to Q. Both values, k and k', are taken right before the load transfer, that is, at the end of a phase. Obviously, P's load was at least 8T at the beginning of the phase, and P can have serviced up to T tasks during the current phase. Finally, P transfers 4T tasks to Q, which leaves it with at least $8T - T - 4T = 3T$ tasks after the transfer, hence $k - k' \geq 3T$. Since Q has received tasks, its load at the beginning of the phase was at most T, and it can have generated up to another T tasks during the phase, leaving it with at most 2T tasks directly before the transfer.

Now we distinguish which queue of P task T came from. First, consider the case $t \in P_{\mathcal{B}}^{(1)}$. We compare the number of tasks with precedence over t on P in system $\mathcal{A}$ with the number of tasks with precedence over t on Q in system $\mathcal{B}$ (after the transfer, not that there is no transfer in system $\mathcal{A}$, so t remains on P in this system). All tasks of P and Q in system $\mathcal{A}$ are moved to their second queues, $P_{\mathcal{A}}^{(2)}$ and $Q_{\mathcal{A}}^{(2)}$. In system $\mathcal{B}$, t is moved from $P_{\mathcal{B}}^{(1)}$ to $Q_{\mathcal{B}}^{(3)}$, and everything from $P_{\mathcal{B}}^{(1)}$ is moved to $P_{\mathcal{B}}^{(2)}$, and everything from $Q_{\mathcal{B}}^{(1)}$ to $Q_{\mathcal{B}}^{(2)}$.

Now server Q of system $\mathcal{B}$ has had at most 2T tasks before the transfer, all of which have precedence over T when it arrives there. Since there are k' tasks with precedence over t that also are transferred, there are at most $2T + k'$ tasks on Q after the transfer with precedence over t in system $\mathcal{B}$.

In system $\mathcal{A}$, at the same time there are at least $3T + k'$ tasks with precedence over t (on server P in queue $P_{\mathcal{A}}^{(2)}$, since t never left P in system $\mathcal{A}$).

The other two cases, $t \in P_{\mathcal{B}}^{(2)}$ and $t \in P_{\mathcal{B}}^{(3)}$, can be handled analogously. We

can conclude, that if some tasks is moved, on its "new" server there are no more tasks with precedence over it than would have been on its "old" server without balancing. This implies that any task's waiting time is not larger in the balancing system, and, hence, the system load also is not larger. □

3.2.3 A Single Phase

In this section, we focus on a single phase of algorithm AlgStochMultiColl. We first place bounds on the numbers of heavy and light servers respectively (Lemma 3.6). Then, we show that the collision games are able to determine a valid assignment from light servers to heavy ones (Lemma 3.9 on page 47).

Estimating the Numbers of Heavy and Light Servers

Lemma 3.6 places bounds on the numbers of heavy and light servers, respectively. First, we introduce a variant of the well-known Chernov Bounds. A proof can be found, among others, in [HR89].

Lemma 3.5 (Chernov bounds I)
Let $n \in \mathbb{N}$ *and let* $p_1, p_2, \ldots, p_n \in \mathbb{R}$ *with* $0 < p_i \leq 1$ *for* $i = 1, \ldots, n$. *Set* $p = p_1 + p_2 + \cdots + p_n$.
Let $X_1, X_2, \ldots, X_n$ *be independent Bernoulli* $\{0, 1\}$ *random variables with* $\text{prob}(X_i = 1) = p_i$ *for* $i = 1, \ldots, n$ *and* $X = X_1 + X_2 + \cdots + X_n$. *Then we have* $\mathrm{E}[X] = p$ *and*

$$\text{prob}(X \geq (1+\epsilon) \cdot p) \leq \begin{cases} \left(\frac{e^{\epsilon}}{(1+\epsilon)^{1+\epsilon}}\right)^{p} & \epsilon \geq 0 \\ e^{\frac{-\epsilon^2 \cdot p}{3}} & 0 \leq \epsilon \leq 1. \end{cases}$$

Now we are ready to formulate the main lemma of this section.

Lemma 3.6 (Heavy and light servers)
Consider an arbitrary but fixed phase. Let $\alpha \geq 1$ and $\mu > 1$ be constants as in Lemma 3.1. Let $\ell_s n$ be an upper bound on the total system load with $\ell_s = \Theta(1)$ (cf. Lemma 3.4).
With a probability of at least $1 - 1/n^{\alpha+1}$, there are

1. *at most $3/(2\log^2 n)$ servers classifying themselves as heavy, and*
2. *at least $n \cdot (1 - (\ell_s/T))$ servers classifying themselves as light.*

Proof. We transform the problem of bounding the number of heavy servers in the balancing system to the problem of estimating the number of servers in the non-balancing system having a certain (other) load.

Recall that a server classifies itself as heavy if its load exceeds the heavy threshold of $8T$ at the beginning of the phase, where T is the length of the phase. What can have happened to this server that forced it to do this classification in the current phase (which we will refer to as phase Π_h from now on)? Suppose that this server has been light during some previous phase Π_ℓ, implying that its load was at most T. Further suppose that it has been selected to be balancing partner for some other (heavy) server during this previous phase, and that it has been assigned another $4T$ tasks due to a balancing action. Finally, during this previous phase it can have generated up to another T tasks on its own, leaving it with at most $T + 4T + T$ tasks at the end of this previous phase. Now for this server to become heavy, it *must* have raised its load on its own by another $2T$ tasks somewhere between phase Π_ℓ and phase Π_h (note that during this interval of time, it may well have been selected to be balancing partner more than once, but that this happens only if its load was less that T at the beginning of the corresponding phases, so the argument still is valid).

Now the probability for a server to increase its load on its own by at least $2T$ tasks in our balancing system clearly can be upper bounded by the probability for a server to have a load of at least $2T$ in the plain non-balancing system. Lemma 3.1 provides probability bounds for this event: with a probability of at most $(1/\mu)^k$, a server has a load of at least k, where the constant $\mu > 1$ is defined as in Lemma 3.1.

Plugging in our values and defining X_i, $1 \leq i \leq n$, to be the load of the

i-th server in the non-balancing system, we obtain

$$\text{prob}\,(X_i \geq 2T) \leq \left(\frac{1}{\mu}\right)^{2T} \leq \frac{1}{\log^2 n}.$$

with $T = 1/10(\log\log n)^2$. Now define another set of random variables Y_i, $1 \leq i \leq n$, where

$$Y_i = \begin{cases} 1 & \text{if } X_i \geq 2T \\ 0 & \text{otherwise} \end{cases}$$

Clearly, $\text{prob}\,(Y_i = 1) = \text{prob}\,(X_i \geq 2T) \leq 1/\log^2 n$. Now define $Y = \sum Y_i$ to be the number of servers in our non-balancing system with a load of at least $2T$. We have $E[Y] \leq n/\log^2 n$, and a simple application of the Chernov bound from Lemma 3.5 yields

$$\text{prob}\left(Y \geq \frac{3}{2} \cdot \frac{n}{\log^2 n}\right) \leq e^{-\frac{n}{12\log^2 n}} \leq \frac{1}{n^\alpha}$$

which proves the first statement of Lemma 3.6.

A pigeonhole argument provides the bound on the number of light servers. We assume a complete system load of at most $\ell_s n$ for some constant ℓ_s (holding with a probability of at least $1 - 1/n^{\alpha+1}$). Clearly, there can be no more that $\ell_s n/T$ servers with a load of at least T. Hence there are at least $n - \ell_s n/T = n(1 - \ell_s/T)$ servers with a load of at most T, which proves the second statement of Lemma 3.6.

□

The Assignment Sub-Phase

Now we are ready to tackle the main part of analyzing a single phase of algorithm ALGSTOCHMULTICOLL. The proof is structured as follows. First, we will show that all the collision games we play to construct the query trees are indeed successful (Lemma 3.7). Then we show that the length of a phase ($T = 1/10(\log\log n)^2$ steps) actually suffices to play these collision games and to perform the additional computation which is required by the algorithm (Lemma 3.8). Finally, we show that given the query trees, we can

successfully find an assignment from light servers to heavy ones (Lemma 3.9 on page 47).

We start by showing that the collision games succeed in building the query trees.

Lemma 3.7 (Collision games)
Let $\alpha \geq 1$, let $a = 4(\alpha + 3)$, $b = a/2$, and let $\epsilon \leq \frac{1}{2(2a)^{a-b+1}}$.
With a probability of at least $1 - 1/n^{\alpha+1}$, each of the $t = \frac{\log\log n}{\log b}$ $(n, \epsilon, a, b, 1)$-collision games successfully terminates after $r = \frac{\log\log n}{\log(b)} + 2$ rounds of its **For***-loop (cf. Figure 2.2 on page 27).*

Proof. Lemma 2.2 shows that an $(n, \epsilon, a, b, 1)$-collision game with a probability of at least $1 - 1/n^{\alpha+2}$ resolves up to ϵn requests in at most r rounds, if

1. $a - b = 2(\alpha + 2) + 2 = 2(\alpha + 3)$, and
2. $\epsilon \leq \frac{1}{2(2a)^{a-b+1}}$, and
3. $r \geq \frac{\log\log n}{\log(a-b)} + 2$.

We now show that all these points are indeed fulfilled.

1. Fulfilled due to our choice of the parameters $a = 4(\alpha+3)$ and $b = a/2$ as stated in the lemma.
2. We need to show that in no round of the assignment sub-phase we have more than ϵn simultaneous requests. For $1 \leq i \leq t$, the number of requests issued in the i-the round of the assignment sub-phase is equal to the number of nodes of the i-th level of the query trees (summarised over all the trees). Consequently, the number of simultaneous requests is maximised in the last round (the collision game determining the structure between the bottom level of the trees and the one above).

   Recall that we develop the trees down to depth $t = \log\log n / \log b$ (by playing as many collision games), and that each internal node has

b "successors". Therefore, the bottom level of each tree consists of at most of $b^{\log\log n/\log b} = \log n$ nodes.

By Lemma 3.6 we know that there w.h.p. are at most $\frac{3n}{2\log^2 n}$ heavy servers each of which a query tree has to be developed for. Hence, in no round of the assignment sub-phase we have more than

$$\frac{3n}{2\log^2 n} \cdot \log n = \frac{3n}{2\log n}$$

requests to be handled simultaneously. This value can be upper bounded by ϵn for any constant $\epsilon > 0$, which implies that the second requirement is fulfilled.

3. We play each collision game for

$$\frac{\log\log n}{\log a - b} + 2 = \frac{\log\log n}{\log b} + 2$$

rounds $(a = 2b)$. Hence, the third requirement is also fulfilled.

Each single collision game succeeds with a probability of at least $1 - 1/n^{\alpha+2}$. Hence, with a probability of at least $1 - 1/n^{\alpha+1}$, all of our $\log\log(n)/\log(b)$ collision games succeed, which proves the lemma. □

Now we show that our phase length of $T = 1/10(\log\log n)^2$ steps is sufficient to successfully play the $\log\log(n)/\log(b)$ collision games and to perform the additional computation which is required by algorithm AlgStochMultiColl.

Lemma 3.8 (Phase Length)
Under the conditions of Lemma 3.7, a phase length of $T = 1/10(\log\log n)^2$ steps is sufficient to perform all the computations made by algorithm AlgStochMultiColl during one phase.

Proof. We play $t = \log\log n/\log b$ collision games, each with $r = \log\log n/\log(b) + 2$ rounds (by definition of our communication model, each round of a collision game takes just one step). We have to additionally charge each collision game with two more steps: one for sending offer

messages, and one for receiving messages whether or not to continue the protocol (whether or not the server initiating the search already has found a partner). Therefore,

$$\frac{\log\log n}{\log b} \cdot \left(\frac{\log\log n}{\log b} + 4\right) \leq \frac{1}{10}(\log\log n)^2 \qquad \text{for } \log b \geq \tfrac{7}{2}.$$

steps are sufficient for the assignment sub-phase. □

Now we are ready to state the main lemma of this section.

Lemma 3.9 (Assignment)
Under the conditions of Lemma 3.7, with a probability of at least $1 - 1/n^{\alpha+1}$, every heavy server get assigned a light server of the phase, whereas each light server is assigned to at most one heavy server.

Proof. The statement that each light server is assigned to at most one heavy server directly follows from the formulation of algorithm ALGSTOCH-MULTICOLL(refer to Section 3.1), since when a light server is designated to become balancing partner its state changes to *non-available* for the rest of the current phase.

Recall that we develop our query trees down to depth $\log\log n/\log b$, and that all the $b^{\log\log n/\log b} = \log n$ nodes on the bottom level of any query tree are pairwise distinct (more, the nodes on the bottom level of the complete query forest are pairwise distinct). We now can proceed and estimate the probability for a successful assignment. To do this, as far as light servers are concerned, we

1. neglect all the nodes of query trees not belonging to the bottom levels, we
2. consider the fact that a single collision game draws a random permutation (a level of the query forest constitutes a random permutation of all nodes) instead of providing truly independent random choices, and we

3. furthermore assume the situation at the end of the assignment sub-phase, where a maximum number of light servers already has been assigned to heavy ones, and which are, therefore, no longer available for other heavy servers still searching. Clearly, the number of heavy servers serves as an upper bound on the number of light servers already assigned to heavy ones.

There are at most $3an/2\log n$ queries issued concurrently during any round of the collision games. If we reduce the number of light servers by this amount, then effects due to the non-independent choices of the collision games are eliminated. Further, recall that we lower bounded the number of light servers by $n(1-\ell_s/T)$ for some constant ℓ_s, and that we upper bounded the number of heavy servers by $3n/(2\log^2 n)$. Hence, at the end of the assignment-sub-phase there are at least

$$n\left(1-\frac{\ell_s}{T}\right)-\frac{3an}{2\log n}\frac{3n}{2\log^2 n}$$

available light servers remaining. Since $T = 1/10(\log\log n)^2$ and therefore $\log^2 n = \Omega(T)$ and also $\log n = \Omega(T)$, we can conclude that the number of still available light servers can be lower bounded by

$$n\left(1-\frac{2\ell_s}{T}\right)=n\left(1-\frac{20\ell_s}{(\log\log n)^2}\right)$$

As mentioned above, the bottom level of any query tree consists of $\log n$ pairwise distinct nodes. Therefore, we can place an upper bound on the probability for the event that there is *no* still available light servers to be found there by

$$\left(\frac{20\ell_s}{\log\log^2 n}\right)^{\log n}\leq\left(\frac{1}{n}\right)^{\alpha+2}$$

for any constant $\alpha > 0$. Hence, with a probability of at least $1-1/n^{\alpha+1}$, each heavy server finds a light one. □

Remark. Assume some server has not classified itself as heavy at the beginning of phase i, implying that its load was at most $8T-1$ (recall that $8T$ is our heavy threshold). Now this server can increase its load by at

most another T tasks during this phase, resulting in a load of $9T - 1$ at the beginning of the next phase. Clearly, this forces this server to declare itself as heavy now. During this next phase, the server can again increase its load by T tasks, leaving it with $10T - 1$ tasks at the end of this phase. Now if a light balancing partner is found successfully, $4T$ tasks are sent to this partner, resulting in a load of $6T - 1$ afterwards.

Furthermore, a server classifies itself as light if its load does not exceed T. It can increase its load by another T tasks during a phase, and can have its load increased by $4T$ tasks due to a load transfer. Therefore, its load is at most $6T$ at the end of the phase.

Finally, a neutral server has a load of at most $8T$ at the beginning of the phase, and can have generated another T tasks, leaving it with at most $9T$ tasks.

Summing up, we see that no server exceeds the maximum allowed load of $10T = (\log \log n)^2$.

3.2.4 Proving the Main Theorem

So far we have focused on a single phase of balancing algorithm AlgStoch-MultiColl. Now we are ready to actually bound the probability for the event of a server exceeding the maximum allowed load of $(\log \log n)^2$ at an arbitrary fixed point of time. The basic idea here is that if such an event actually takes place, then there must have been a phase in which this server's attempt at a balancing action has failed (it has classified itself as heavy but has not found a balancing partner). First we need to again estimate the complete system load, but now for a phase from which we assume that there actually *is* a server classifying itself as heavy.

Lemma 3.10 (System load in balancing system II)
Given the conditions of Lemma 3.4 on page 39, suppose that at the beginning of phase Π some server P classifies itself as heavy and has not done so the phase before. Then, with a probability of at least $1 - 1/n^{\alpha+1}$, the system load still can be bounded by $\ell_s n$ (with ℓ_s being some constant).

Proof. Let

- A be the event that there are more than $\ell_s n$ tasks in the system at the first time step of phase Π, and
- B be the event that P classifies itself as heavy.

Clearly,

$$\text{prob}(A|B) = \frac{\text{prob}(A \cap B)}{\text{prob}(B)} \leq \frac{\text{prob}(A)}{\text{prob}(B)}.$$

Closely following Lemma 3.3, we can show that $\text{prob}(A) \leq 1/n^{\alpha+2}$ for any constant α and ℓ_s as given in Lemma 3.4.

Now that we have an upper bound on $\text{prob}(A)$, we derive a lower bound on $\text{prob}(A)$. For this, consider the following, modified system.

1. The phase length now is $8T = 8/10(\log\log n)^2$ instead of T.
2. The load generation and servicing distribution is identical to the original system, as well as the classification sub-phase.
3. After the classification sub-phase,
   - each non-heavy server deletes all tasks it currently keeps and changes classification to light if it has not been before, and
   - each heavy server deletes all but the $4T$ tasks it is going to transfer to a balancing partner.

Obviously, there are fewer load transfers in this system than in the original one. Now we lower bound the probability for a fixed server of this, modified system to become heavy. To become heavy (from previously non-heavy), a server must have increased its load from zero to $8T$ during the last phase (every server deletes all its tasks at the beginning of a phase – except for tasks to be transferred, which are deleted at the first step of the next phase). According to Lemma 3.1 on page 35, the probability for this event is

$$\left(p^{(+)}\right)^{8T} = (p \cdot (1 - (p(1+\epsilon))))^{4/5(\log\log n)^2} \geq \frac{1}{n}$$

for any constants $0 < p < 1$ and $\epsilon > 0$ such that $p(1+\epsilon) < 1$. Since now the probability for a server to become heavy is only larger in the original system, we can conclude that $\text{prob}(B) \geq 1/n$, and, consequently,

$$\text{prob}(A|B) \leq \frac{\text{prob}(A)}{\text{prob}(B)} \leq \frac{n}{n^{\alpha+2}} = \frac{1}{n^{\alpha+1}}$$

□

As a consequence of this lemma, we can conclude that even under the assumption of a server changing its classification to heavy at the beginning of some phase, we know that all the lemmas which rely on an $\mathcal{O}(n)$ bound on the system load still are valid.

We are ready to prove Theorem 1.1 (on page 18) now. We upper bound the probability for the existence of a server P exceeding the maximum allowed load of $(\log\log n)^2$ at some arbitrary time step τ. Let $\Pi(\tau)$ be the phase step τ belongs to. Clearly, there must be some previous phase Π' in which server P changed its classification from non-heavy to heavy, and during the consecutive phases $\Pi', \ldots, \Pi(\tau)$ server P always was heavy, meaning that P either did not find a balancing partner at all, or that it did find one but was not able to transfer sufficiently many tasks in order to drop below the heavy threshold at the beginning of the next phase.

Let z denote the phase number of phase $\Pi(\tau)$. Clearly, $\Pi' = \Pi(\tau) - i$ for some $1 \leq i < z$. Now we define three events:

- $\mathcal{E}_i$ is the event that the classification of P changed the last time from not heavy to heavy in the first step of phase $\Pi(\tau) - i$, $1 \leq i < z$ (this means $\Pi' = \Pi(\tau) - i$).
- $\mathcal{F}$ is the event that P has still a load larger than $(\log\log n)^2$ in step t (belonging to phase $\Pi(\tau)$).
- $\mathcal{U}$ is the event that the first balancing attempt of P in phase Π' has been unsuccessful.

We now have

$$\begin{aligned}
&\text{prob}\left(\text{P has a load larger than } (\log\log n)^2 \text{ in step } t\right) \\
&\leq \sum_{i=1}^{z-1} \text{prob}\left(\mathcal{F} \wedge \mathcal{E}_i\right) \\
&\leq \sum_{i=1}^{z-1} \text{prob}\left(\mathcal{E}_i\right) \cdot \text{prob}\left(\mathcal{F} \mid \mathcal{E}_i\right)
\end{aligned}$$

The condition that the first balancing attempt was unsuccessful is *necessary* for our situation of P being overloaded in step t. It follows that $\text{prob}(\mathcal{F} \mid \mathcal{E}_i) \leq \text{prob}(\mathcal{U})$, and

$$\begin{aligned}
&\text{prob}\left(\text{P has a load larger than } (\log\log n)^2 \text{ in step } t\right) \\
&\leq \sum_{i=1}^{z-1} \text{prob}\left(\mathcal{E}_i\right) \cdot \text{prob}\left(\mathcal{U}\right) \\
&= \text{prob}\left(\mathcal{U}\right) \cdot \sum_{i=1}^{z-1} \text{prob}\left(\mathcal{E}_i\right) = \text{prob}\left(\mathcal{U}\right)
\end{aligned}$$

We know that the first balancing attempt of a fixed heavy server P succeeds with a probability of at least $1 - 1/n^{\alpha+1}$, even if we know that in this phase P changes its classification to heavy. Hence,

$$\text{prob}\left(\mathcal{U}\right) \leq \frac{1}{n^{\alpha+1}}.$$

It directly follows that we can upper bound the probability for *any* server exceeding the maximum allowed load of $(\log\log n)^2$ by $1/n^{\alpha}$, and Theorem 1.1 follows.

3.2.5 Other Generation Models

This section is about Theorem 1.2, stating a $\Delta \cdot (\log\log n)^2$ upper bound on the load of any server at any arbitrary but fixed point of time for balancing algorithm AlgStochMultiColl, given the StochGeometric(Δ) and the StochBinomial(Δ, p) generation model, respectively.

The fact that the bound is slightly worse than the one from Theorem 1.1 for generation model StochBernoulli(p, ϵ) is due to the situation that

now a server may generate up to Δ tasks per step. Hence, during a phase of length T it can generate up to ΔT tasks, and ΔT is an inherent lower bound on what can be shown to be the maximum load of any server.

In order to be able to apply algorithm AlgStochMultiColl to generation models StochGeometric(Δ) and StochBinomial(Δ, p), we need to slightly modify the algorithm (this has to be done to account for the influence of Δ, as described above). We have to scale our thresholds by a factor of Δ (actually, if we had parameterised the algorithm with Δ, no change at all would have been necessary, but might have complicated the discussion in the previous sections):

1. A server classifies itself as heavy, if its load exceeds $8\Delta T$ at the beginning of a phase (was $8T$ in the original algorithm).
2. A server classifies itself as light, if its load is less than ΔT at the beginning of a phase (was T in the original algorithm).
3. During a balancing action, $4\Delta T$ tasks are transferred from heavy servers to light ones (was $4T$ in the original algorithm).

The next lemma shows that we can establish bounds on the load of single servers as we have done for generation model StochBernoulli(p, ϵ) in Lemma 3.1 on page 35.

Lemma 3.11 (Load of a single server II)
Let $\Delta \geq 1$ be an arbitrary integer constant, and let $p \in (0, 1)$ such that $\Delta p < 1$.

Given the StochGeometric(Δ) *or the* StochBinomial(Δ, p) *generation model, there is a constant $\mu > 1$, such that with a probability of at most $(1/\mu)^k$, a server will have a load of at least k at any arbitrary but fixed point of time in a non-balancing system.*

Proof. To prove this lemma, we rely on some mechanisms of Queueing Theory, Although mostly concerned with continuous time problems, these mechanisms and approaches are applicable to our inherent discrete time problem.

When observing a bulk-arrival system with individual service, it is a well-known concept in Queueing Theory to treat the bulks as "super customers" (with expected service time equal to the sum of the expected service times of the single customers belonging to the bulk) and therefore to obtain a simple birth-death-process, see, for instance, [Coh82, Kle96].

If we model our single servers in a non-balancing system as standard queues, we have an interarrival time of bulks of constant one per queue, and a service time for single customers of also constant one (obviously, in our case a bulk is the set of tasks generated by a server in one time step). If we now combine these tasks to form a "super task", then we have to set its expected service time to the expected bulk size (which depends on the generation model) times the expected service time of the single customers (which in our case is constant one).

There is a nice way to prove an exponential decrease in the waiting time distribution of customers in a GI/G/1 queue (see [Kin64]). Since our model with super tasks fits into this model, we are going to apply this mechanism. Assume the tasks being generated by a server at the same time step to belong to a super task. Now

1. let s_n denote the service time of the n-th super task,
2. let t_n denote the time between the arrivals of the n-th and the $(n+1)$-th super task, and
3. let $u_n = s_n - t_n$.

In [Kin62] it is shown that if $E[u_n] < 0$ then the waiting time distribution converges. In [Kin64] is is shown that if the expectation of the moment generating function Φ for u_n exists for some real $\theta > 0$, that is,

$$E[\Phi(\theta)] = E[e^{\theta u_n}] = \sum_{x:\ \mathrm{prob}(u_n = x) > 0} e^{\theta x} \cdot \mathrm{prob}\,(u_n = x) < \infty,$$

then $\Phi(\theta) \leq 1$ and furthermore, with w denoting the waiting time in equilibrium,

$$\mathrm{prob}\,(w_n \geq z) \leq \mathrm{prob}\,(w \geq z) \leq e^{-\vartheta z}$$

for $\vartheta = \sup\{\theta > 0 | \; \Phi(\theta) \le 1\}$ if the queue initially is empty.

Generation models StochGeometric(Δ) and StochBinomial(Δ, p) allow for at most a constant number of tasks to be generated per time step. In both cases the expected number of tasks generated per server and time step is less than one, and the interarrival time of super tasks as well as the service time of single tasks is constant one. Hence, our super tasks have service time s distributed in $[0, \Delta]$ with $E[s] < 1$. As $u_n = s_n - t_n$ and $t_1 = t_2 = \cdots = 1$, we have

$$
\begin{aligned}
& \sum_{x:\ \mathrm{prob}(u_n = x) > 0} e^{\theta x} \cdot \mathrm{prob}\,(u_n = x) \\
= & \sum_{x:\ \mathrm{prob}(s_n - t_n = x) > 0} e^{\theta x} \cdot \mathrm{prob}\,(s_n - t_n = x) \\
= & \sum_{x:\ \mathrm{prob}(s_n - 1 = x) > 0} e^{\theta x} \cdot \mathrm{prob}\,(s_n - 1 = x) \\
= & \sum_{x:\ \mathrm{prob}(s_n = x + 1) > 0} e^{\theta x} \cdot \mathrm{prob}\,(s_n = x + 1) \\
= & \sum_{x=-1}^{\Delta - 1} e^{\theta x} \cdot \mathrm{prob}\,(s_n = x + 1) \\
= & \sum_{x=0}^{\Delta} e^{\theta (x-1)} \cdot \mathrm{prob}\,(s_n = x) \\
\le & \; e^{\theta(\Delta - 1)} \cdot \sum_{x=0}^{\Delta} \mathrm{prob}\,(s_n = x) \\
= & \; e^{\theta(\Delta - 1)}.
\end{aligned}
$$

Since Δ is a constant, $e^{\theta(\Delta - 1)} < \infty$ for all $\theta > 0$. Now by [Kin64], there exists θ such that $\Phi(\theta) \le 1$, and

$$\mathrm{prob}\,(w \ge z) \le e^{-\vartheta z}$$

for $\vartheta = \sup\{\theta > 0; \; \Phi(\theta) \le 1\}$. Finally, for constant $\nu = e^{\vartheta} > 1$ we have for w,

$$\mathrm{prob}\,(w \ge z) \le \left(\frac{1}{e}\right)^{\vartheta z} = \left(\frac{1}{\nu}\right)^{z}.$$

Since there is a one-to-one correspondence of waiting time and queue length in our model (all tasks have a service time of constant one), and a super task consists of at most Δ regular tasks, we can easily find a $\mu = \mu(\nu, \Delta) > 1$ such that for the queue length ℓ, $\text{prob}(\ell \geq z) \leq 1/\mu^z$, and the lemma is proven. □

Now that we know that we can again estimate the load of a single server in a non-balancing system, we can apply the same lemmas as we have done in the previous sections of this chapter:

1. Bound the complete system load both in the non-balancing system and the balancing one.
2. Place bounds on the numbers of heavy and light servers, respectively.
3. Show that the that the collision games succeed.
4. Finally, show that the we actually find an assignment from light to heavy servers, and show that the phase length is sufficient.

3.2.6 Recovery Properties

Of course, if our system is run long enough, then it will approach any (degenerated) state, i.e., a state in which the system load cannot be bounded by $\mathcal{O}(n)$. Fortunately, it has the inherent property of "automatical recovery" back to some typical state, where the bound on the system load again is valid.

We can restrict ourselves on showing this property for a non-balancing system, since the main essence of Lemma 3.4 is that the consumption rate in a balancing system is not smaller than in a non-balancing system. Another hint on this is that in worst-case scenarios, our balancing algorithm just does *nothing*; be it that the collision games do not succeed, or that no valid assignment from light servers to heavy ones can be determined.

Now consider a non-balancing system in some arbitrary state, let ℓ denote the maximum load of any server. Obviously, in any interval of length $r\ell$,

a server expectedly generates $r\ell p$ tasks, and expectedly services $r\ell p(1+\epsilon)$ tasks. Now for $r \geq 1/(\epsilon p)$, we have that

$$\ell + r\ell p \leq r\ell p(1+\epsilon),$$

implying, that given an interval large enough (depending on ℓ, p, and ϵ), the servers not only expectedly service all the tasks they have generated during this interval, but also those present at the beginning of the interval.

CHAPTER 4

The Balancing Algorithm ALGSTOCHSINGLECOLL

In this chapter we introduce and analyse the second load balancing algorithm, ALGSTOCHSINGLECOLL, for the stochastical load generation scheme. Where algorithm ALGSTOCHMULTICOLL of the previous Chapter 3 needs a phase length of $\Theta((\log\log n)^2)$, implying an inherent lower bound of $\Theta((\log\log n)^2)$ on the maximum load of any server, for algorithm ALGSTOCHSINGLECOLL now a phase length of $\Theta(\log\log n)$ suffices. We will show that the maximum load of any server can be upper bounded by $\Theta(\log\log n)$, which is an asymptotic decrease of a factor $\log\log n$.

Algorithm ALGSTOCHSINGLECOLL again follows the sketch from Section 1.4. Compared to algorithm ALGSTOCHMULTICOLL, the assignment subphase has changed significantly; actually, it has been completely redesigned. The main difference is that, where algorithm ALGSTOCHMULTICOLL played $\mathcal{O}(\log\log n)$ collision games, one after the other, algorithm ALGSTOCHSINGLECOLL now plays just one such game during the assignment subphase (that's where the names come from).

In the remainder of this chapter, we first introduce balancing algorithm ALGSTOCHSINGLECOLL in Section 4.1, and then analyse its performance in Section 4.2.

SECTION 4.1

The Algorithm

Time is divided into consecutive phases of length $T = \ell \log\log n$ for some constant ℓ to be specified later. Since we again assume a stochastical load generation model (STOCHBERNOULLI(p, ϵ) throughout most of the chapter, where a server generates a task with a probability of p per step, and services a task with a probability of $p(1+\epsilon)$), there is no need for the load estimation sub-phase, because we again can exploit the inherent $\mathcal{O}(n)$ upper bound on the complete system load. A phase of algorithm ALGSTOCHSINGLECOLL now looks as follows.

Classification. Servers having a load of at most T at the beginning of the phase are classified as *light*, and servers with a load of at least $8T$ at the beginning of the phase are classified as *heavy*. The remaining servers are classified as *neutral*. Note that the thresholds are not with respect to some "floating" value but fixed and only dependent on T.

Assignment. Again, we want to find an assignment from light servers to heavy ones. In algorithm ALGSTOCHMULTICOLL, we established the so-called query trees, and every heavy server sought for a light balancing partner in its own tree (which, as we have seen, might overlap with other servers trees). Now we do not play $\log\log n$ collision games in order to establish this "search structure" but just one. This sub-phase is split in two main parts.

First, we play a collision game where all the servers issue requests. Imagine the accepting of queries as throwing directed edges into a graph which initially has had no edges at all. Thus, when a server issues a request consisting of a queries, of which b are accepted, this means that there are b directed edges from this server to the servers having accepted the queries. Obviously, playing a $c = 1$ collision game is out of question; one has to find parameters that allow to show that the collision game actually is able to resolve the requests, w.h.p. (as we will see, $a = 2\alpha + 5$, $b = a - 1$, and $c = 8ea^2$ are just fine). To summarise: the collision game builds a directed graph (with the servers as nodes) where each node has an out-degree of

b and an in-degree of at most c, w.h.p. The servers having accepted the queries of another server P are called P's successors, and P is the predecessor of these servers. Note that we identify servers and nodes in this context; we will switch notations quite frequently, and use whatever seems more appropriate.

This graph G then is used to find the assignment. Again, the heavy servers make use of tree-like structures embedded in G – just not explicitly constructed level by level. For any server P, let $T_P \subset G$ consist of P itself as "root" and all the nodes reachable by directed paths of a length of at most $t = \Theta(\log\log n)$ (will be specified later). Note that again several T_P can overlap, or that some node can occur more than once in one T_P.

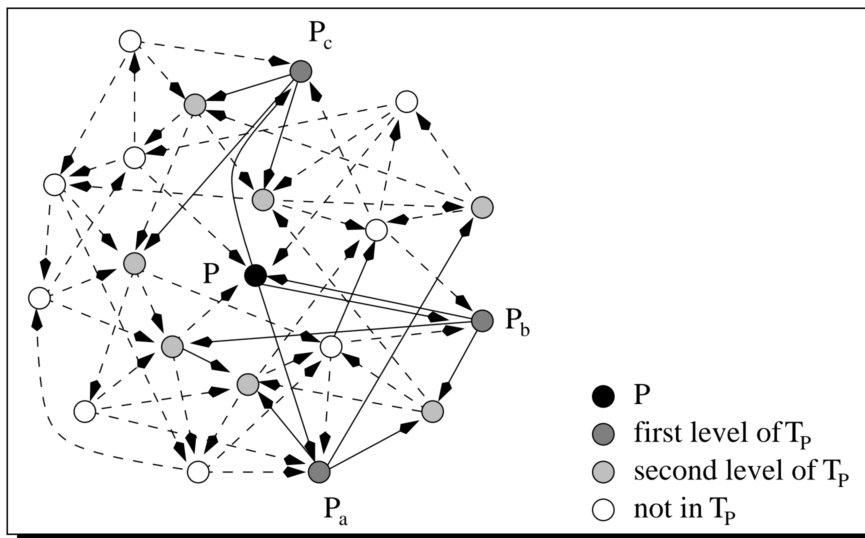

Figure 4.1: An example T_P ($b = 3$, $c = 4$, depth $= 2$).

Figure 4.1 shows a typical outcome of a collision game which has been played with parameters $b = 3$ and $c = 4$. Node P in the centre (black) has three successors P_a, P_b, and P_c (dark grey), which themselves each have three successors (light grey). Dashed lines represent choices of the collision game (accepted queries), and solid lines indicate affiliation to our 3-level T_P. Note that some nodes are successor to more than one node, for instance P_a and P_b share a common successor (which then shows up twice on level two of the T_P), and node P_b has P itself as successor, which then occurs on level zero and two.

Now every heavy server P tries to find a balancing partner in its T_P. This works as follows.

1. Every heavy server sends *search messages* to its b successors. Such a message initially consists of nothing but the heavy server's ID. Now for $t-1$ rounds, each server having received such a search message (or more than one of them – recall that the in-degree of any node may be as large as $c = 8ea^2$) in the previous round, combines them if necessary, and forwards them to its own successors. It additionally stores the IDs of the corresponding servers in a list of its own. Note that our communication model allows for a constant number of messages to be sent/received per step.

2. After these t rounds, each light server sorts its list and compacts it by deleting dupes (now in each such list every server occurs at most once). Recall that we do not assume any explicit communication network underlying the servers. Since most reasonable networks have a diameter of $\mathcal{O}(\text{polylog}(n))$ (and, hence, routing of messages takes as long), we allow for $\mathcal{O}(\text{polylog}(n))$ computational steps per communication step.

3. Now every light server randomly selects one ID of its compacted list (uniformly distributed). It then sends an *offer message* to the corresponding server. In turn, a heavy server having received one or more offer messages, selects an arbitrary one of them and answers with an *acknowledgement message* to this server, which is now designated as its balancing partner. Note again that our communication model allows for this, since at most a constant number of messages have to be evaluated per server, regardless of how many arrive simultaneously.

Transfer. Each heavy server transfers $4T$ of its tasks to its balancing partner.

SECTION 4.2

The Analysis

In this section we analyse the performance of algorithm AlgStochSingleColl given load generation model StochBernoulli(p, ϵ). The analysis is split into the same parts as the one for algorithm AlgStochMultiColl in Section 3.2.

1. First we again need to derive an upper bound on the complete system load. Since the system itself is just the same as before, there is no need to modify the analysis. Hence, assuming load generation scheme StochBernoulli(p, ϵ), Lemma 3.1 on page 35, Lemma 3.3 on page 37, and Lemma 3.4 on page 39 are still valid, implying that
   (a) with a probability of at most $1/\mu^k$, a server will have a load of at least k at any arbitrary but fixed time step in the non-balancing system, where $\mu > 1$ is some constant depending on p and ϵ, and
   (b) with a probability of at least $1 - 1/n^{\alpha+1}$, the complete system load can be bounded by $\mathcal{O}(n)$, both in the non-balancing system and in the balancing one (note that the proof of Lemma 3.4 does in no way depend on the length of a phase, and that everything else remains unchanged, as far as the proof is concerned).
2. Next, in Section 4.2.1, we analyse a single phase of our algorithm. We provide an upper bound on the number of heavy servers as well as a lower bound on the number of light servers. We show that the collision game actually resolves all the requests, and that the assignment from light servers to heavy ones succeeds (with high probability).
3. In Section 4.2.2, we prove the main theorem of this part, namely Theorem 1.4 on page 19. This is done similar to the proof of Theorem 1.1 for algorithm AlgStochMultiColl.
4. Finally, in Section 4.2.3, we again consider the two other stochastical load generation models, namely StochBinomial(Δ, p) and StochGeometric(Δ).

Note that the recovery property we have stated in Section 3.2.6 also still is valid, since it depends on the load generation only and not on the actual algorithm.

4.2.1 A Single Phase

The structure of this section closely follows the one of Section 3.2.3, where we analysed a single phase of algorithm AlgStochMultiColl. Since there is no need to estimate the complete system load again (Lemma 3.4 still is valid), we can start right away with providing bounds both on the number of heavy servers as well as on the number of light servers (Lemma 4.1). Then, we show in Lemma 4.5 that we indeed find a valid assignment from light to heavy servers within out assignment sub-phase (w.h.p.).

Estimating the Numbers of Heavy and Light Servers

The following lemma provides bounds on the numbers of heavy and light servers, respectively.

Lemma 4.1 (Heavy and light servers)
Consider an arbitrary but fixed phase of length $T = \ell \log\log n$ *for some constant* ℓ. *Let* $\alpha \geq 1$ *and* $\mu > 1$ *be constants as in Lemma 3.1.*
For a fixed server, the probability for it to be heavy is at most $1/\mu^{2\ell \log\log n}$, *and there are at least* $\frac{3}{4}n$ *light servers.*

Proof. The proof follows the outline of the proof of Lemma 3.6, where we have done almost the same for algorithm AlgStochMultiColl. We now just have to take care of the different phase length.

Recall that a server is heavy if its load is at least 8T at the beginning of this phase. This means that it somehow must have raised its load on its own by at least 2T (light with at most T tasks at the beginning of some phase, plus another T self generated tasks during this previous phase, plus 4T tasks due to a load transfer leaves it with at most 6T tasks — 2T tasks are "missing" in order to become heavy).

Now the probability for a server to have a load of 8T in our balancing system can be upper bounded by the probability for it to have a load of at

least $2T$ in a non-balancing system. Following Lemma 3.1, we can upper bound this probability by

$$\left(\frac{1}{\mu}\right)^{2T} = \left(\frac{1}{\mu}\right)^{2\ell \cdot \log\log n}$$

with $\mu = \frac{\epsilon+(1-p-\epsilon p)}{1-p-\epsilon p} > 1$.

It remains to lower bound the number of light servers. According to Lemma 3.4 (which clearly still is valid, as there are absolutely no references to the length of a phase in its proof), we can bound the compete system load by $\ell_s n$ for some constant ℓ_s. Obviously, there cannot be more than $\ell_s n/T$ servers with a load of at least T (since then the system load would have to be larger). Hence, there are at least

$$n - \frac{\ell_s n}{T} = n \cdot \left(1 - \frac{\ell_s}{T}\right) = n \cdot \left(1 - \frac{\ell_s}{\ell \log\log n}\right) \geq n \cdot \left(1 - \frac{1}{4}\right) = \frac{3}{4}n$$

servers with a load of at most T (the light ones), which finishes the proof of the lemma. □

The Assignment Sub-Phase

Now we are ready to do the main part of analysing a single phase of algorithm AlgStochSingleColl. The proof is structured as follows. First, we will show that the collision game we play to construct the access structure (containing all the T_P's) is indeed successful (Lemma 4.2). Next, in Lemma 4.3, we provide a result concerning a key property of this access structure. Then, we show that given the T_P's, we can successfully find an assignment from light servers to heavy ones (Lemma 4.5). Finally, we show that the length of a phase ($T = \ell \log\log n$ steps for some constant ℓ) actually suffices to play this game and to perform the additional computation which is required by the algorithm (Lemma 4.6).

Recall that we play one collision game with parameters $a = 2\alpha + 5$, $b = a - 1$, and $c = 8ea^2$. Proving that the collision game actually succeeds is a straightforward application of Lemma 2.1.

Lemma 4.2 (Collision game)
Let $\alpha \geq 1$ *be an arbitrary constant, let* $a = 2\alpha + 5$, $b = a - 1$, $c = 8ea^2$, *and* $\epsilon = 1$.
With a probability of $1 - 1/n^{\alpha+1}$, *the* (n, ϵ, a, b, c)*-collision protocol is able to resolve the* n *requests within* $\frac{\log\log n}{\log c} + 1$ *rounds of the* **For**-*loop.*

Proof. Just plug the parameters into Lemma 2.1. □

Now we are going to explore some structural properties of our access graph and the T_P's. A heavy server tries to find a balancing partner in its T_P. For this reason, we would like a T_P to contain as many pairwise distinct nodes as possible. Recall that a node may occur more than once in a T_P – even worse, if it occurs twice, the complete subtrees rooted in it are identical!

Fix some T_P. Let $s_\ell(T_P)$ denote the *logical size* of T_P, that is, counting all the occurrences of the nodes, no matter if they occur once, twice, or more often (that is, assuming a regular b-ary tree). Since we have an out-degree of b and the T_P's have a depth of t,

$$s_\ell(T_P) = \frac{b^{t+1} - 1}{b - 1} \leq 2b^t.$$

Now let $s_p(T_P)$ denote the *physical size* of T_P. Here, we want to count just one occurrence of any node of T_P, i.e., we are interested in the size of the maximum cardinality subset of pairwise distinct nodes. Figure 4.1 on page 61 shows an example of a T_P with $b = 3$ and three levels. The logical size clearly is $1 + 3 + 9 = 13$, but due to some nodes occurring more than once (consider, for instance, the common successor of P_a and P_b), the physical size reduces to 11 (all the non-white nodes). The following lemma provides a lower bound on the physical size of any T_P.

Lemma 4.3 (Physical size)
Consider T_P*'s with out-degree* $b = 2\alpha + 4$ *of depth* $t = t' \log\log n$ *for some constant* t'.
With a probability of at least $1 - 1/n^{\alpha+1}$,

$$s_p(T_P) \geq \frac{b-2}{b} \cdot s_\ell(T_P) \geq \frac{3}{4} \cdot s_\ell(T_P)$$

for every T_P.

Proof. Fix an arbitrary T_P. In order to find the desired bound, we apply a *deletion process* to T_P. This process traverses all the nodes of T_P in breadth-first-search (BFS) order, and it deletes nodes which correspond to servers that already have occurred earlier during the traversal. In more detail, the process for some node v looks as follows.

Deletion process for node v.

If the server which is represented by v has not occurred earlier during the BFS traversal of T_P, then do nothing. Otherwise, delete the complete subtree rooted in v (including v), and mark the edge from v to its predecessor as *cut-off edge*. Of course, there is no *real* edge from v to its father, because v is no longer there. Just imagine this edge as "virtual".

Note that we never actually do this in the algorithm; it's just a theoretical means of finding a lower bound on $s_p(T_P)$. Anyway, repeat this until either there are no more nodes there to be visited, or there are at least $\ell = b/2+1$ cut-off edges. In this case we apply a *final deletion step* in which we delete all nodes that have not yet been visited.

Each T_P consists of the root P and b subtrees of height $t-1$ each. Since the (a, b, c) collision game chooses a different targets for any request (see Figure 2.2), the children of any node v represent pairwise distinct servers, and there is at most one child of v representing the same server as v itself does. Hence, in the worst case, one cut-off edge deletes one of the direct subtrees of the root P, and the remaining ones cut the tree the level below (cutting a subtree rooted in a successor of a successor of P).

Since there are at most $b/2$ cut-off edges at all if we do not have to face the final deletion step, then in the worst case we have to delete one subtree of height $t-1$ and $b/2-1$ subtrees of height $t-2$ each. A straightforward calculation reveals that, for $b \geq 8$, we have at least

$$\frac{b^{t+1}-1}{b-1} - \frac{b^t-1}{b-1} - \left(\frac{b}{2}-1\right) \cdot \frac{b^{t-1}-1}{b-1}$$
$$\geq \frac{b-2}{b} \cdot \frac{b^{t+1}-1}{b-1} \geq \frac{3}{4} \cdot \frac{b^{t+1}-1}{b-1} = \frac{3}{4} \cdot s_\ell(T_P)$$

nodes left. It now remains to bound the probability that a final deletion step occurs for some fixed T_P. Let $\{e_1 = (u_1, v_1), \ldots, e_k = (u_k, v_k)\}$ denote

the set of cut-off edges, and let r_i denote the request that has been issued by the server represented by node u_i, $1 \leq i \leq k$. Hence, to each request r_i belongs a query that is directed to a server which is represented by a node that already has been visited during the BFS traversal before. Recall that the logical size of T_P is

$$\begin{aligned}
\frac{b^{t+1}-1}{b-1} &\leq 2b^t \\
&\leq 2b^{t' \log\log n} \\
&= 2b^{t' \log(b) \log\log(n)/\log(b)} \\
&= 2 \cdot 2^{t' \log(b) \log\log(n)} \\
&= 2 \cdot (\log n)^{t' \log b}
\end{aligned}$$

Now we are ready to compute the sought probability.

1. There are at most $(2 \cdot (\log n)^{t' \log b})^{b/2+1}$ ways to choose the $b/2+1$ cut-off edges whose existence lead to a final deletion step.

2. For each cut-off edge e_i we can choose one of the $2 \cdot (\log n)^{t' \log b}$ nodes that represent the same server and have been visited before during the BFS traversal.

3. The probability that one of the a queries belonging to request r_i is directed to w_i is no more than a/n.

Therefore, the probability for a final deletion step is at most

$$\begin{aligned}
& (2 \cdot (\log n)^{t' \log b})^{b/2+1} \cdot (2 \cdot (\log n)^{t' \log b})^{b/2+1} \cdot (a/n)^{b/2+1} \\
= & \left(\frac{4 \cdot (\log n)^{2t' \log b} \cdot a}{n}\right)^{b/2+1} \leq \left(\frac{1}{n}\right)^{\alpha+2}
\end{aligned}$$

for $b \geq 2(\alpha+2)$. Hence, with a probability of at least $1 - 1/n^{\alpha+1}$, $s_p(T_P) \geq 3/4 \cdot s_\ell(T_P)$ for every T_P. This finishes the proof of the lemma. □

Before we can proceed and bound the probability for a success of the assignment sub-phase we need to introduce another Chernov-like tail estimate.

Lemma 4.4 (Chernov bounds II)
Let $n \in \mathbb{N}$ and let $p_1, \ldots, p_n \in \mathbb{R}$ with $0 < p_i \leq 1$ for $i = 1, \ldots, n$. Let $p = \sum p_i$. Let $X_1, \ldots, X_n$ be independent Bernoulli random variables with $\text{prob}(X_i = 1) = p_i$ for $i = 1, \ldots, n$, and let $X = \sum X_i$. Then, $\mathrm{E}[X] = p$, and

$$\text{prob}(X \leq (1-\epsilon) \cdot p) \leq e^{-\epsilon^2 p/2}.$$

for $0 \leq \epsilon \leq 1$

A proof can be found in [HR89]. We now can state the following lemma.

Lemma 4.5 (Assignment)
Let $\alpha \geq 1$ be an arbitrary constant, let $a = 2\alpha + 5$, $b = a - 1$, $c = 8ea^2$, and let $k = 8(\alpha + 2)^2$. Let $\mu > 1$ be defined as in Lemma 3.1. Let $t = (2 \log\log(n) + \log(k)) / \log b$ be the depth of the T_P's. Suppose a phase has a length of length of $T = \ell \cdot \log\log n$ with $\ell \geq \frac{1}{\log \mu} \cdot \left(\frac{3}{2} + \frac{\log c}{\log b}\right)$. With a probability of at least $1 - 1/n^{\alpha+1}$, every heavy server gets assigned a light server at the end of the assignment sub-phase, and every light server is assigned to at most one heavy server.

Proof. First, note that here the formula describing the depth of the T_P's is a little bit more complicated than in the formulation of Lemma 4.3. What we have here is the "true depth", whereas in Lemma 4.3 a simplification was sufficient.

The second statement clearly follows directly from the formulation of algorithm AlgStochSingleColl.

A fixed T_P has logical size of

$$\begin{aligned}
\frac{b^{t+1} - 1}{b - 1} &\geq b^t \\
&= b^{(2 \log\log(n) + \log(k))/\log b} \\
&= 2^{2 \log\log(n) + \log(k)} \\
&= 2^{2 \log\log n} \cdot 2^{\log k} \\
&= (\log n)^2 \cdot k \\
&= (\log n)^2 \cdot 8(\alpha + 2)^2
\end{aligned}$$

By Lemma 4.3 from above, with a probability of at least $1 - 1/n^{\alpha+1}$, the physical size is at least half this value, hence

$$s_p(T_P) \geq (\log n)^2 \cdot 4(\alpha + 2)^2.$$

By Lemma 4.1, at least $3/4 \cdot n$ servers are light. Due to the fact that the collision game restricts the in-degree of any node to at most c, its choices are not independent. However, similar to the proof of Lemma 3.9, by reducing the number of light servers by the size of a T_P, we can overcome this problem. Since $|T_P| = \mathcal{O}(\text{polylog}(n)) < n/4$, we can conclude that the probability for a fixed server determined by the collision game is light, is at least $3/4 - 1/4 = 1/2$.

Let X_P denote the expected number of light servers in T_P. By the above,

$$E[X_P] \geq 1/2 \cdot s_p(T_P) \geq (\log n)^2 \cdot 2(\alpha + 2)^2.$$

Application of the Chernov tail estimate from Lemma 4.4 and straightforward calculation yields

$$\text{prob}\left(X_P \leq (\alpha + 2)^2 \cdot (\log n)^2\right) \leq \left(\frac{1}{n}\right)^{\alpha+2}.$$

for any constant $\alpha > 1$. Hence, with a probability of at least $1 - 1/n^{\alpha+1}$, every T_P contains at least $(\alpha + 2)^2 \cdot (\log n)^2$ pairwise distinct light servers.

Now we bound the number of heavy servers P that contain some fixed light server P' in their T_P (these are the servers that P' receives a search message from, and that P' might send an offer message to). We say that P' is *reachable* by these servers.

As any node of our access graph has an in-degree of at most c, we can upper bound the number of servers which can reach P' by $(c^{t+1} - 1)/(c - 1) \leq 2c^t$ (recall that any T_P has a depth of t). By Lemma 4.1, the probability for a randomly chosen server to be heavy is at most $1/\mu^{2\ell \log\log n}$. With Y_P as the expected number of heavy servers that can reach some light server P, we obtain $E[Y_P] \leq 2c^t \cdot 1/\mu^{2\ell \log\log n}$.

We now want to choose ℓ such that

$$2c^t \left(\frac{1}{\mu}\right)^{2\ell \log\log n} \leq \log n.$$

with $t = (2\log\log n + \log k)/\log b$. Taking logarithms and solving for ℓ yields

$$\begin{aligned} & 2c^t\left(\frac{1}{\mu}\right)^{2\ell\log\log n} \le \log n \\ \iff\ & 1 + t\log c - 2\ell\log\log n\cdot\log\mu \le \log\log n \\ \iff\ & \ell \ge \frac{1+t\log c - \log\log n}{2\log\log n\log\mu} \end{aligned}$$

If we now replace t with its value, this yields

$$\begin{aligned} \ell &\ge \frac{1+t\log c - \log\log n}{2\log\log n\log\mu} \\ &= \frac{1+\frac{2\log\log n+\log k}{\log b}\cdot\log c - \log\log n}{2\log\log n\log\mu} \\ &= \frac{\log b + (2\log\log n + \log k)\cdot\log c - \log b\cdot\log\log n}{2\log\log n\log\mu\log b} \\ &= \frac{1}{2\log\log n\log\mu} + \frac{\log c}{\log\mu\log b} + \frac{\log k\log c}{2\log\log n\log\mu\log b} - \frac{1}{2\log\mu} \\ &= \frac{1}{\log\mu}\cdot\left(\frac{1}{2\log\log n} + \frac{\log c}{\log b} + \frac{\log k\log c}{2\log\log n\log b} - \frac{1}{2}\right). \end{aligned}$$

Since obviously the terms

$$\frac{1}{2\log\log n} \quad \text{and} \quad \frac{\log k\log c}{2\log\log n\log b}$$

tend to 0 for growing n, we can safely replace them with 1 each, and

$$\ell \ge \frac{1}{\log\mu}\cdot\left(2 + \frac{\log c}{\log b} - \frac{1}{2}\right)$$

is a sufficient condition for

$$E[Y_P] \le 2c^t\left(\frac{1}{\mu}\right)^{2\ell\log\log n} \le \log n.$$

Now we can apply Chernov bounds (Lemma 3.5) to obtain

$$\text{prob}\left(Y_P \ge (\alpha+2)\log n\right) \le \frac{1}{n^{\alpha+2}}$$

Hence, with a probability of at least $1-1/n^{\alpha+1}$, each light server is reachable by at most $(\alpha+2)\log n$ heavy servers.

Recall that in algorithm AlgStochSingleColl, light servers randomly select one of the heavy servers they have the ID of in their list, and then send an offer message to these selected servers. With high probability, for any light server P, there are at most $(\alpha+2)\log n$ heavy servers which can reach P. Hence, the probability that a heavy server is chosen by a light one is at least one is at least $1/((\alpha+2)\log n)$.

Now fix a heavy server P. We know that w.h.p. T_P contains at least $(\alpha+2)^2\cdot(\log n)^2$ pairwise distinct light servers. Hence, the probability that P is selected by none of these light servers is at most

$$\begin{aligned} & \left(1-\frac{1}{(\alpha+2)\log n}\right)^{(\alpha+2)^2(\log n)^2} \\ = & \left[\left(1-\frac{1}{(\alpha+2)\log n}\right)^{(\alpha+2)\log n}\right]^{(\alpha+2)\log n} \\ \leq & \left(\frac{1}{e}\right)^{(\alpha+2)\log n} \leq \left(\frac{1}{n}\right)^{\alpha+2} \end{aligned}$$

It follows that with a probability of at least $1-1/n^{\alpha+1}$, every heavy server is selected by at least one light server. □

In the previous lemma we have derived a lower bound on the length of a phase of $T=\ell\log\log n$ with $\ell\geq\frac{1}{\log\mu}\cdot\left(\frac{3}{2}+\frac{\log c}{\log b}\right)$ (this length was necessary to ensure that the assignment works). It remains to show that this bound also is sufficient as fast as the time is concerned that is needed to play the collision game, the basic building block of the assignment sub-phase, and to pass the messages up and down the T_P's.

Lemma 4.6 (Phase length II)
Under the conditions of Lemma 4.5, a phase length of $T=\ell\log\log n$ steps is sufficient to perform all the computations made by algorithm AlgStochSingleColl during one phase, and to ensure that the assignment sub-phase is successful.

Proof. Clearly playing the collision game is one of the most time consuming part of any phase. By Lemma 4.2, this takes $\log\log(n)/\log(c)+1$ steps (recall that, due to our communication model, we can run a round of the collision game during a step).

As far as the messages are concerned, a T_P has a depth of $t = (2\log\log n + \log k)/\log b$, and clearly sending the search messages takes t steps.

Not considering a few (constant) additional steps (classification, transfer, handling of offer/acknowledgement messages, and the like), we see that we need to have a phase length of at least

$$\begin{aligned} & \frac{\log\log n}{\log c} + \frac{2\log\log n + \log k}{\log b} \\ \leq\ & \frac{\log\log n}{10} + \frac{2\log\log n + \log k}{2} \\ =\ & \frac{\log\log n}{10} + \log\log n + \frac{\log k}{2} \\ \leq\ & 2\log\log n \end{aligned}$$

since $b = 2\alpha+4 \geq 6 \Rightarrow \log b \geq 2$, $c = 8ea^2 = 8e(2\alpha+5)^2 \geq 1024 \Rightarrow \log c \geq 10$, and $k = 8(\alpha+2)^2$ is constant. Hence, a phase length of at least $2\log\log n$ is sufficient for performing a phase of algorithm AlgStochSingleColl.

Lemma 4.5 places another bound on the minimum length of a phase, namely $\ell \log\log n \geq \frac{1}{\log\mu}\left(\frac{3}{2} + \frac{\log c}{\log b}\right)\log\log n \geq 2\log\log n$. Therefore, a phase length of $\ell\log\log n$ satisfies both conditions. □

Remark. Again we can spend a few words on what happens if the first balancing attempt of a server (right after having classified itself as heavy and the phase before not) is successful.

The load of a non-heavy server is at most $8T-1$. It then can have increased by at most T during a phase, leaving it with at most $9T-1$ tasks (the classification now switches to heavy). Again, T tasks can be generated, resulting in at most $10T-1$. The transfer reduces by $4T$, and the final load is at most $6T-1$ (note: again not heavy). Similarly, light servers (at most T tasks) can not become heavy due to balancing transfers.

4.2.2 Proving the Main Theorem

This section is quite similar to Section 3.2.4, in which we have shown the validity of the main theorem for algorithm AlgStochMultiColl. Again, we bound the probability for the event that at any arbitrary but fixed time step there is a server exceeding the maximum allowed load of $10T$.

The first thing we have to do is to again bound the complete system load at the beginning of some phase of which we already know that there is a server changing its classification from formerly non-heavy to heavy.

Lemma 4.7 (System load in balancing system III)
Given the conditions of Lemma 3.4 on page 39, suppose that at the beginning of phase Π some server P classifies itself as heavy and has not done so the phase before. Then, with a probability of at least $1 - 1/n^{\alpha+1}$, the system load still can be bounded by $\ell_s n$ (with ℓ_s being some constant).

Proof. The proof of this lemma is almost entirely identical to the one for Lemma 3.10, just that we have to cope with different phase lengths. Recall that we have defined two events, namely A the event of the system load being at least $\ell_s n$, and B the event that server P initiates a balancing action and has not done so the phase before. Clearly, $\text{prob}(A|B) \leq \text{prob}(A)/\text{prob}(B)$. We can show $\text{prob}(A) \leq 1/n^{\alpha+2}$ and with the same reasoning as in the proof of Lemma 3.10 we can conclude that $\text{prob}(B) \geq 1/n$, resulting in $\text{prob}(A|B) \leq 1/n^{\alpha+1}$. □

Now we are ready to finish the proof of Theorem 1.4. We know that we can bound the system load (in a non-balancing system as well as in a balancing one – even if we know that there is a heavy server), we know that the collision game succeeds, that a proper assignment from light to heavy servers will be found, and that the phase has a sufficient length.

The remainder of the proof again is similar to the one of Theorem 1.1. We upper bound the probability that a server exceeds the maximum allowed load of $10T$ at some arbitrary but fixed time step t by the probability that its first attempt at a balancing action (with it being heavy from the time of this first attempt until time step t) failed. According to the observations

from above, we can bound this probability by $1/n^{\alpha+1}$, and, hence, the probability for any server exceeding this value is at most $1/n^{\alpha}$. This finishes the proof of Theorem 1.4.

4.2.3 Other Generation Models

As we have done for algorithm AlgStochMultiColl in Section 3.2.5, we now investigate the behaviour of algorithm AlgStochSingleColl for the two other stochastical load generation model, StochGeometric(Δ) and StochBinomial(Δ, p). We obtain the same result: the maximum load increases by just the constant factor of Δ which specifies how many tasks a server may generate per step (recall that in either case, the expected number of tasks generated is less than one).

We again have to slightly modify the algorithm in that we have to adjust the heavy and light thresholds as well as the number of tasks transferred during a balancing action. All three values have to be multiplied with Δ. (Note that if we had parameterised the algorithm accordingly, then no change at all would be necessary.)

The main difference to the "standard" analysis assuming generation scheme StochBernoulli(p, ϵ) is that we have to carefully estimate the queue lengths of single servers under generation models StochGeometric(Δ) and StochBinomial(Δ, p). Once this is done (with Lemma 3.11, using techniques from Queueing Theory), then the whole machinery which has been used to prove Theorem 1.4 on page 19 (for StochBernoulli(p, ϵ)) can again be used to now prove Theorem 1.5 on page 19 (for StochGeometric(Δ) and StochBinomial(Δ, p)), without the need for any further change: the bound on the load of any single server in a non-balancing system implies a bound on the complete system load, which, in turn, allows us to prove that the collision game succeeds, and that an assignment from light to heavy servers is possible (and indeed found). We can conclude that we again can upper bound the probability for a server to exceed the maximum allowed load of now $10\Delta T$ at any arbitrary but fixed time step t by the probability of a failure of its first balancing attempt (the first in a continuous row ending at time step $t - \text{now}$). This finishes the proof of Theorem 1.5.

CHAPTER 5

The Balancing Algorithm ALGADV

With this chapter we leave the stochastical load generation schemes we have investigated throughout the previous two chapters and turn our attention over to our adversarial schemes. Recall that now there is no longer some probabilistic distribution determining the generation and consumption of tasks, but an adversary is allowed to change the load of any server by at most some constant Δ per step (or, equivalently, to change its load by at most ΔT in an interval of $T = (\log \log n)^2$ steps).

This chapter is structured like the previous ones: we first introduce the balancing algorithm ALGADV in Section 5.1, and then analyse its performance in Section 5.2.

SECTION 5.1 The Algorithm

As usual, the algorithm follows the sketch from Section 1.4 on page 12. We divide time into phases of length $T = 2(\log \log n)^2$. The fact that now an adversary is allowed to specify when and where tasks are generated and serviced, respectively, has two main consequences:

1. We cannot place an upper bound on the complete system load as has been possible for the stochastical schemes (the adversary could decide to just generate tasks and never service them).

2. It no longer makes sense to try and bound the maximum load of any server. Hence, our analysis concentrates on bounding the difference between the maximum load of any server and the average system load.

Obviously, classifying servers as heavy or light without regarding the the average system load, also no longer makes sense. Hence, we now need to provide the servers with some information before having them classifying themselves. This is what the load estimation sub-phase is for. We let the servers just estimate the average system load instead of computing the real value because this would be much too time consuming. In the following, we present algorithm ALGADV in more detail.

Load estimation. Recall the access graph we made use of for algorithm ALGSTOCHSINGLECOLL of Chapter 4, and the sub-graphs T_P the heavy servers tried to find a light balancing parter in. For the load estimation sub-phase we will exploit the very same structure. We play an (a, b, c)-collision game with parameters $a = 2\alpha + 5$, $b = a - 1$, and $c = 8ea^2$, where every server issues a request. This results in a random directed graph on n nodes, where each node has an out-degree of b (edges to servers that have accepted one of the queries belonging to its request) and an in-degree of at most c. For a server/node P (we again identify servers and corresponding nodes in this graph), again let T_P be defined as the subgraph consisting of P as root and of all the nodes reachable by directed paths of a length of at most $t = (2 \log\log(n) + \log(k))/\log(b)$, with $k = 8(\alpha + 2)^2$, and $\alpha > 1$ the exponent of the polynomial probability bound.

Now instead of searching for a balancing partner in T_P, every server P computes the average load of all the servers in its T_P. Although a T_P's size is just polylog(n), we will see that this serves as a usable estimation for the complete system load. Every server maintains a list of tuples $(i, \ell_i + 6\Delta T)$, where i denotes the identifier of some server, and ℓ_i denotes this servers load at the first time step of the current phase. We will use two different notations: if we refer to a server as P, its load will be denoted as ℓ_P, and if we refer to it by its ID i, $1 \leq i \leq n$, then its load will be denoted as ℓ_i. It is just most important that ℓ_P or ℓ_i always refers to the load of the corresponding server at the first time step of the current phase. We will refer of the term $\ell_i + 6\Delta T$ as the i-th servers *normalised load.* Initially, for

each server P_i, $1 \le i \le n$, this list consists of just $(i, \ell_i + 6\Delta T)$, that is, its own identifier and its own normalised load.

The load estimation now works as follows.

1. First, each server sends its list consisting of one tuple up to its predecessors in the access graph, if there are any predecessors (there are at most c of them).

2. Then, for $t - 1$ rounds, each server merges the lists it has received from its successors to obtain a sorted list (sorted by the identifiers), and deletes any dupes. This results in a sorted list in which any servers tuple occurs at most once. It then sends this list up to all its predecessors in the access graph.

3. Obviously, after these t rounds, a server P has gathered the load information (based on the very first time step of the current phase) from all the servers in its T_P, because it has received information from all the servers which are *reachable* from P, according to the definition of reachability of Chapter 4. Additionally, each member of T_P contributes exactly one entry to the list.

   Now, each server P computes the *normalised average load* $\bar{\ell}_P$ of its T_p by summing up the normalised load values contained in the tuples and by dividing this value by the size of the list (the number of items).

Recall that our model allows for complex operations as merging two lists, since its timing model is communication step based. We assume a diameter of the underlying network of polylog(n), and hence allow for polylog(n) steps of computation per step of communication.

Classification. Servers P having a load ℓ_P of at most $3\bar{\ell}_P$ at the beginning of the phase are classified as *light*, and servers with a load of at least $(6/\epsilon) \cdot \bar{\ell}_P$ at the beginning of the phase are classified as *heavy*. The remaining servers are classified as *neutral*. ϵ is the parameter of the $(n, \epsilon, a, b, 1)$ collision games we are going to play during the assignment phase (right now it suffices to know that $\epsilon \in (0, 1)$ is a constant).

Note that this classification takes place after $\Theta(\log \log n)$ steps, and that the decisions are based on the load situation of the first time step of he current phase.

Assignment. Unfortunately, for determining an assignment from light servers to heavy ones, we cannot make use of the approach of the previous chapter, where a heavy server P tried to find a balancing partner in its T_P. This is due to the fact that now we cannot bound the number of heavy servers as nicely as we could for algorithm ALGSTOCHSINGLECOLL — as we will see, now we can show that there is at most a constant fraction of all servers heavy, whereas for ALGSTOCHSINGLECOLL, this was $n/\text{polylog}(n)$. Using this approach, this constant fraction is by far too much if we want to make sure that all the heavy servers find a partner. Fortunately, we can switch back to the approach of algorithm ALGSTOCHSINGLECOLL, where we develop a "query tree" for each heavy server by playing $\Theta(\log\log n)$ collision games with collision parameter $c = 1$, one after the other.

Now we play $t = \frac{\log\log n}{\log b} + \alpha + 1$ $(n, \epsilon, a, b, 1)$ collision games with $a = 4(\alpha + 2)$, $b = 2\alpha + 4$, and $\epsilon = \cdot(2a)^{-(a-b+1)}$ (we may not allow for more than ϵn simultaneous requests per collision game). The assignment subphase now works as follows.

1. As in ALGSTOCHMULTICOLL, in the first collision game only heavy servers issue requests. If one or more of the b servers having accepted a query belonging to a request is light, then they send *offer messages* to the heavy server that has issued the request.Note that, due to $c = 1$, a server accepts at most one query, and, consequently, sends at most one offer message. Every heavy server having received an offer message arbitrarily selects one of them and sends back an *acknowledgement message*. The corresponding server now is designated as balancing partner, and changes its state to *non-available*. Furthermore, each heavy server sends either a *stop searching message* to its successors if it actually has found a partner already, or it sends a *continue searching message* otherwise.

2. Now each server having accepted a query during the previous, first collision game itself issues a request in a new $(n, \epsilon, a, b, 1)$ collision game if and only if it has received a continue searching message the step before — its predecessor, a heavy server, has not yet found a partner. The ID of the predecessor is passed along with the queries.

   Note that we have to stop the search for a balancing partner as soon as possible, because, unlike in ALGSTOCHMULTICOLL, we now have

that a constant fraction of all servers could be heavy, and developing all query trees down to maximum depth of t would be impossible (this will become clear in the analysis).

Each light server now having accepted a query of this second collision game, and that still is available (not yet selected to be balancing partner), sends an offer message to the heavy server whose ID was passed along with the query, and this server, in turn, arbitrarily selects one of them and returns an acknowledgement message (which leads to a change in state to non-available). It also sends a continue searching message or a stop searching message to its direct successors (depending on whether it has or has not found a partner now), and this message is passed down to the successors of the successors (which have accepted a query in this second collision game).

This procedure continues for $t-2$ rounds, where the ID passed along with the query always is the one of the heavy server that has issued a request in the very first collision game, of course (the "owner" of this query tree).

It should be noted, that although all servers on a level of a query tree (and even on a level of the complete query forest) are pairwise distinct, a server may well occur on more than one level of a tree (meaning that it's only a tree if we do not identify the two nodes corresponding to the same server). This reduces the "real" number of servers in a query tree, but if this tree is developed to depth t, then the bottom level contains sufficiently many pairwise distinct servers, as we will see in the analysis.

Further note that since a server P can occur on up to $\mathcal{O}(\log\log n)$ levels of the query forest (just not more than once on the same level), this results in a memory requirement of $\mathcal{O}(\log(n)\log\log(n))$ bits, since P has to store the identifiers of the b servers ($\log n$ bits each) that have accepted a query of P's request in order to be able to forward the stop/continue searching messages properly.

Transfer. Heavy servers P transfer $3/\epsilon \cdot \bar{\ell}_P$ tasks to their designated light balancing partners.

Note that for the same reason that we could not make the heavy and light

thresholds independent of the current load situation, we now have to take it into account for the number of tasks to be transferred. Further, note that both the thresholds and the number of transferred tasks vary from server to server (due to the differing estimations of the average load).

SECTION 5.2

The Analysis

In this section we analyse the performance of algorithm AlgAdv. We assume load generation model AdvInterval(Δ) for this purpose, but all statements are true for model AdvStep(Δ) as well. This section is structured as follows.

1. In Section 5.2.1 we analyse the load estimation sub-phase. We show that the collision game succeeds in resolving the requests, and we place bounds on the deviation of the estimation of the average load from the true average load for any server.
2. In Section 5.2.2, we analyse the assignment sub-phase. First, we place bounds on the numbers of heavy and light servers. Then, we show that the collision games succeed, and that indeed a valid assignment from light to heavy servers determined.
3. In Section 5.2.3 we show that if the system is in a balanced state at the beginning of one phase, then it w.h.p. remains so not only for the next phase, but for the next poly(n) phases.
4. Finally, we examine two minor aspects of algorithm AlgAdv. In Section 5.2.4 we analyse its expected communication overhead, and in Section 5.2.5 we propose some modifications which make it resistant to worst-vase scenarios.

Before we can jump into the analysis, we have to make a few definitions. Fix some phase Π (of length $r(\log\log n)^2$ for some constant r to be specified later), and let τ be the first time step of phase Π. Let M_τ denote the complete system load at time step τ, let

$$m_\tau = \frac{M_\tau}{n} + 6\Delta T$$

denote the *normalised average load.* Let

$$M'_\tau = \sum_{i=1}^{n} (\ell_i + 6\Delta T) = n \cdot (6\Delta T) + \sum_{i=1}^{n} \ell_i = n \cdot (6\Delta T) + M_\tau.$$

We then have

$$\frac{M'_\tau}{n} = \frac{n \cdot (6\Delta T) + M_\tau}{n} = 6\Delta T + \frac{M_\tau}{n} = m_\tau$$

We call the system λ-*balanced* for a given phase Π and some constant $\lambda \geq 1$, if at the first time step τ of Π, no server has a load exceeding

$$U_\lambda = \lambda \cdot \left(\frac{M_\tau}{n} + 12\Delta(\log\log n)^2 \right).$$

Since $T = 2(\log\log n)^2$, we have $12\Delta(\log\log n)^2 = 6\Delta T$, and, by definition of m_τ,

$$U_\lambda = \lambda \cdot \left(\frac{M_\tau}{n} + 6\Delta T \right) = \lambda m_\tau.$$

5.2.1 The Load Estimation Phase

In this section we analyse the load estimation sub-phase of algorithm ALG-ADV. The analysis is structured as follows.

1. In Observation 5.1 we show that the collision game succeeds in resolving the n requests, and thus establishes our access graph, in which the servers calculate their estimation of the average load.

2. Lemma 5.3 then shows that these estimations are indeed quite useful,that is, the estimations are within a constant factor of the true average load.

Observation 5.1 (Collision game)
We play the (a, b, c) collision game with the same parameters as in the assignment sub-phase of algorithm ALGSTOCHSINGLECOLL, *namely $a = 2\alpha + 5$, $b = a - 1$, and $c = 8ea^2$. Hence, by Lemma 4.2, which is just a straightforward application of Lemma 2.1, with a probability of at least $1 - 1/n^{\alpha+1}$, the collision game will be successful in resolving the n requests.*

Before we can continue and show that the estimations of the average load are good, we need to introduce another tail estimate, known as *Azuma's Martingale Tail Estimate.* Its nice property is that the random variables in question do not have to be independent (unlike in standard Chernov bounds).

Lemma 5.2 (Azuma's Martingale Tail Estimate)
Let $X_0, X_1, X_2, \dots$ *be a Martingale sequence such that for every* $k > 0$ *and* c *independent of* k,

$$|X_k - X_{k-1}| \leq c.$$

Then, for all $t \geq 0$ *and any* $\lambda \geq 0$,

$$\text{prob}\left(|X_t - X_0| \geq \lambda c \sqrt{t}\right) \leq 2 \exp(-\lambda^2/2).$$

We now place bounds on the quality of the load estimations $\bar{\ell}_P$.

Lemma 5.3 (Load estimation)
Fix some phase Π, *and let* τ *be the first time step of* Π. *Under the assumption that the system is* λ*-balanced for phase* Π, *with a probability of at least* $1 - 1/n^\alpha$,

$$\frac{2}{3} \cdot m_\tau \leq \bar{\ell}_P \leq \frac{4}{3} \cdot m_\tau$$

for any server P, *and for any constant* $\alpha > 1$.

Proof. Fix some server P. Its T_P has a depth of

$$t = \frac{2 \log\log(n) + \log(k)}{\log b}$$

with $k = 8(\alpha + 2)^2$. (This is the α from Theorem 1.6, but does not really matter here, any positive constant will do). A T_P with out-degree b has a logical size (counting each and every occurrence of nodes representing servers) of $(b^{t+1} - 1)/(b - 1)$. Let R denote the set of servers occurring in the T_P (without multiple occurrence), let D denote $|R|$, the physical size. By Lemma 4.3 on page 66,

$$\begin{aligned} D &\geq \frac{3}{4} \cdot \frac{b^{t+1}-1}{b-1} \geq \frac{3}{4} \cdot b^t \\ &= \frac{3}{4} \cdot b^{\frac{2\log\log(n)+\log(k)}{\log b}} \\ &= \frac{3}{4} \cdot 2^{2\log\log(n)+\log(k)} \\ &= \frac{3}{4} \cdot (\log n)^2 \cdot k \\ &\geq (\log n)^2. \end{aligned}$$

Now let X denote the random variable counting the sum of the normalised loads of all the servers in R; each server in R occurs just once, and the load is taken at the beginning of the current phase. Now for $0 \leq i \leq D$, define random variables X_i such that X_i is the expected value of X when the load of the first i members of R is known, and the load of the remaining $D-i$ is not. The sequence of X_i forms a Doob martingale sequence, and we have $X_0 = E[X]$, and $X_D = X$.

We now want to apply Azuma's martingale tail estimate on this sequence. In order to do so, we need to upper bound $|X_i - X_{i-1}|$ for $i > 0$ by some value independent of i. Let Y_i be the normalised load of the i-th member of R (recall that the normalised load is defined as the load plus $6\Delta T$). Let

$$S_i = M'_\tau - \sum_{j=1}^{i} Y_j = \sum_{j=1}^{n} (\ell_j + 6\Delta T) - \sum_{j=1}^{i} Y_j,$$

that is, S_i is the complete normalised load of all servers except for the first i members of R.

Since now by assumption we know the normalised load of the first i members of R, and the expected complete normalised load of the remaining $D-i$ members of R is just $D-i$ times the average over *all* servers except these i ones, namely

$$\frac{S_i}{n-i} \cdot (D-i),$$

we can conclude that

$$X_i = \frac{S_i}{n-i} \cdot (D-i) + \sum_{j=1}^{i} Y_j.$$

By assumption the system is λ-balanced for this phase. This implies that at the first step of this phase, the load of any server is at most λm_τ. Hence, for any $i \in \{1, \ldots, D\}$ there is a $j \in \{1, \ldots, n\}$ such that

$$\begin{aligned} Y_i &= \ell_j + 6\Delta T \le \lambda m_\tau + 6\Delta T = \lambda\left(\frac{M_\tau}{n} + 6\Delta T\right) + 6\Delta T \\ &\le 2\lambda\left(\frac{M_\tau}{n} + 6\Delta T\right) = 2\lambda m_\tau, \end{aligned}$$

since $\lambda \ge 1$. Further, since obviously $S_i \le S_{i-1}$, we obtain

$$\begin{aligned} & X_i - X_{i-1} \\ &= \left(\frac{S_i}{n-i}\cdot(D-i) + \sum_{j=1}^{i} Y_j\right) - \left(\frac{S_{i-1}}{n-i+1}\cdot(D-i+1) + \sum_{j=1}^{i-1} Y_j\right) \\ &= \frac{S_i}{n-i}\cdot(D-i) - \frac{S_{i-1}}{n-i+1}\cdot(D-i+1) + Y_i \\ &\le \frac{S_{i-1}}{n-i}\cdot(D-i) - \frac{S_{i-1}}{n-i+1}\cdot(D-i+1) + 2\lambda m_\tau \\ &= S_{i-1}\cdot\left(\frac{D-i}{n-i} - \frac{D-i+1}{n-i+1}\right) + 2\lambda m_\tau \end{aligned}$$

Since $S_{i-1} \ge 0$ and $\frac{D-i}{n-i} < \frac{D-i+1}{n-i+1}$ for $0 \le i \le D < n$, we can conclude

$$X_i - X_{i-1} \le 2\lambda m_\tau.$$

We are now ready to apply Lemma 5.2, and we obtain

$$\begin{aligned} & \operatorname{prob}\left(|X_D - X_0| \ge \frac{1}{3}\frac{M'_\tau}{n}\cdot D\right) \\ &\le 2\exp\left(-\frac{1}{2}\cdot\left(\frac{(M'_\tau/n)\cdot\sqrt{D}}{3\cdot 2\lambda m_\tau}\right)^2\right) \\ &\le 2\cdot\exp\left(-\frac{D}{72\lambda^2}\cdot\left(\frac{M'_\tau/n}{m_\tau}\right)^2\right) \\ &\le 2\cdot\exp\left(-\frac{D}{72\lambda^2}\right) \qquad \left[\frac{M'_\tau}{n} = m_\tau \text{ by definition}\right] \\ &\le 2\cdot\exp\left(-\frac{(\log n)^2}{72\lambda^2}\right) \\ &\le \frac{1}{n^{\alpha+1}}. \end{aligned}$$

for any constant $\alpha > 1$. So, we know that w.h.p.,

$$|X_D - X_0| \leq \frac{1}{3} m_\tau D,$$

Since $X = X_D = \ell_P D$ and $X_0 = E[X] \leq m_\tau D$, it follows that for a fixed server, with a probability of at least $1 - 1/n^{\alpha+1}$,

$$\frac{2}{3} m_\tau D \ \leq \ X = \bar{\ell}_P D \ \leq \ \frac{4}{3} m_\tau D,$$

and, consequently, with a probability of at least $1 - 1/n^{\alpha}$,

$$\frac{2}{3} m_\tau \ \leq \ \bar{\ell}_P \ \leq \ \frac{4}{3} m_\tau,$$

for any server P. This finishes the proof of the lemma. □

5.2.2 The Assignment Sub-Phase

In this section we analyse the assignment sub-phase of algorithm ALGADV. It is structured as follows.

1. Lemma 5.4 places bounds on the numbers of heavy and light servers, respectively.
2. In Lemma 5.5 we show that the collision games played during the assignment sub-phase are able to resolve all the requests, that is, to develop the query trees.
3. In Lemma 5.6 we show that the assignment from light servers to heavy ones indeed succeeds.
4. Lemma 5.7 finally shows that a phase length of $T = 2(\log\log n)^2$ suffices to do all the work (including the load estimation).

We first bound the numbers of heavy and light servers, respectively.

Lemma 5.4 (Heavy and light servers)
Let $\alpha \geq 1$ and $\lambda > 1$ be an arbitrary constant. Let $0 < \epsilon = \frac{1}{2(2a)^{a-b+1}} < 1$ denote the parameter of the collision games played during the assignment sub-phase (the maximum number of simultaneous requests is ϵn). Assume the system to be λ-balanced for the current phase.

With a probability of at least $1 - 1/n^{\alpha}$, there are at most $\frac{\epsilon n}{4}$ heavy servers, and there are at least $\frac{n}{2}$ light servers.
Furthermore, with a probability of at least 3/8, a fixed node represents a light server.

Proof. We prove the two statements by simple pigeonhole arguments. Consider a heavy server P, that is, a server with a load ℓ_P of at least $(6/\epsilon) \cdot \bar{\ell}_P$. Now by Lemma 5.3, $\bar{\ell}_P \geq (2/3)m_\tau$, and hence

$$\ell_P \geq \frac{6}{\epsilon} \cdot \bar{\ell}_P \geq \frac{6}{\epsilon} \cdot \frac{2}{3} m_\tau = \frac{4}{\epsilon} m_\tau$$

Consequently, there can be at most

$$\frac{n}{\frac{4}{\epsilon} m_\tau} = \frac{\epsilon n}{4 m_\tau} \leq \frac{\epsilon n}{4}$$

servers being heavy. On the other hand, a light server P has a load ℓ_P of at most $3\bar{\ell}_P$. This implies (again by Lemma 5.3) that any non-light server Q has a load of at least

$$\ell_Q \geq 3\bar{\ell}_Q \geq 3 \cdot \frac{2}{3} m_\tau = 2m_\tau.$$

Hence, there are at most

$$\frac{n}{2m_\tau} \leq \frac{n}{2}$$

non-light servers, and the first two statements of the lemma follow directly.

For the third statement, note that the choices of the collision game are not truly independent since a $c = 1$ collision game determines an assignment such that any server accepts at most one query (this would not be the case for a random function). Hence, if we assume the worst-case that (a) some node is on the bottom level of its query tree, and (b) that all other nodes on the bottom level of the same query tree also represent light servers (pairwise distinct), then by reducing the probability by b^t/n (b^t is the number of nodes on this level), we have eliminated these dependencies completely.

Additionally, when assuming a node on the bottom level, the number of available light servers decreases since there can be light servers already assigned (and, therefore, non-available when occurring again on the bottom level) on some other level above. Since there are at most $\epsilon n/4$ heavy servers, the number of still available light servers on the bottom level can decrease by at most this number, and the corresponding probability decreases by at most $\epsilon/4$.

All in all, since there are at least $n/2$ light servers, the remaining probability for a fixed server to be light is

$$\frac{1}{2} - \frac{\epsilon}{4} - \frac{b^t}{n} = \frac{1}{2} - \frac{\epsilon}{4} - \frac{b^{\alpha+1} \cdot \log n}{n} \geq \frac{3}{8}.$$

This finishes the proof of the lemma. □

Next we can proceed and show that all the $t = \frac{\log\log n}{\log b} + \alpha + 1$ collision games are able to resolve their requests.

Lemma 5.5 (Collision games)
Let $\alpha \geq 1$ be an arbitrary constant. Let $a = 4(\alpha + 2)$, $b = a/2$, and let $\epsilon = \frac{1}{2(2a)^{a-b+1}}$.
With a probability of at least $1 - 1/n^{\alpha}$, each of the $t = \frac{\log\log n}{\log b} + \alpha + 1$ collision games succeeds in resolving the requests within $\frac{\log\log n}{\log b} + 2$ rounds per game.

Proof. We need to place a bound on the number of simultaneous requests issued per collision game of at most ϵn. We prove this lemma by an induction argument.

Due to the bound on the number of heavy servers from Lemma 5.4, we know that w.h.p., there are at most $\epsilon n/4$ requests issued in the first collision game.

Now fix some round of the assignment sub-phase, and assume that $r \leq \epsilon n$ requests are issued in this round.

If $r \leq \epsilon n/b$, then in the next round, at most $b \cdot (\epsilon n/b) = \epsilon n$ requests are issued, since each requests involves b successful queries, which themselves can issue a request each in the next round.

On the other hand, if $(\epsilon n)/b < r \le \epsilon n$, then define r random variables $X_1, \ldots, X_r$, such that $X_i = 1$ if none of the b queries belonging to the i-th request was accepted by a light and still available server, and let $X_i = 0$ otherwise. Let $X = \sum X_i$. By Lemma 5.4, $\text{prob}(X_i = 1) \le (5/8)^b$ for $1 \le i \le r$. Hence, since $\alpha \ge 1$ and $b = a/2 = 2(\alpha + 2) \ge 6$,

$$E[X] \le \epsilon n \cdot \left(\frac{5}{8}\right)^b \le \frac{\epsilon n}{2b}.$$

Now we can apply Chernov bounds to estimate the deviation ox X from its expected value and obtain

$$\text{prob}\left(X \ge \frac{\epsilon n}{b}\right) \le \frac{1}{n^{\alpha+1}}.$$

Consequently, with a probability of at least $1 - 1/n^{\alpha}$, we have at most ϵn simultaneous requests in any of the collision game, which proves the lemma; just plug the parameters a, b, $c = 1$, and ϵ into Lemma 2.2. □

In the next lemma we show that the assignment sub-phase indeed determines a valid assignment from light to heavy servers.

Lemma 5.6 (Assignment)
Let $\alpha \ge 1$ be an arbitrary constant. Assume the system to be λ-balanced for the current phase.

With a probability of at least $1 - 1/n^{\alpha}$, each heavy server gets assigned a light one during the assignment sub-phase. Furthermore, each light server is assigned to at most one heavy server.

Proof. By Lemma 5.5, we know that with a probability of at least $1-1/n^{\alpha}$, the

$$t = \frac{\log\log n}{\log b} + \alpha + 1$$

collision game succeed in developing the query trees of depth t. Since each node has an out-degree of b, the bottom level of each tree consists of

$$b^t = b^{\log\log n/\log b + \alpha + 1} = b^{\alpha+1} \cdot \log n$$

nodes representing pairwise distinct servers (recall that $c = 1$). By Lemma 5.4, with a probability of at least $3/8$, each of these nodes represents a light and still-available server. Hence, we can bound the probability of the event that for a fixed server the bottom level of its query tree consists of either non-light or light but non-available servers by

$$\left(\frac{5}{8}\right)^{b^t} = \left(\frac{5}{8}\right)^{b^{\alpha+1}\cdot\log n} = \left(\frac{1}{n}\right)^{\log(8/5)\cdot b^{\alpha+1}} \leq \left(\frac{1}{n}\right)^{\alpha+1}.$$

Consequently, with a probability of at least $1 - 1/n^{\alpha}$, each heavy server finds at least one light and still available server on the bottom level of its query tree.

The second statement of the lemma directly follows from the definition of the algorithm (each light server accepts at most once to become balancing partner). □

Finally, we show that the phase length of $2(\log\log n)^2$ indeed is sufficient to play all the collision games of both the load estimation sub-phase and the assignment sub-phase of algorithm AlgAdv (note that the classification sub-phase and the transfer sub-phase take just one step).

Lemma 5.7 (Phase length)
$T = 2(\log\log n)^2$ *steps are sufficient for one phase of algorithm* AlgAdv.

Proof. During the load estimation sub-phase we play one (n, ϵ, a, b, c) collision game with parameters $a = 2\alpha+5$, $b = a-1$, $c = 8ea^2$, and $\epsilon = 1$. By Lemma 2.1,

$$\frac{\log\log n}{\log b} + 1$$

rounds are sufficient to resolve all the request, w.h.p. The T_P's have a depth of

$$t = \frac{2\log\log(n) + \log(8(\alpha+2)^2)}{\log(b)} \leq 2\log\log n,$$

hence gathering the load information from a T_P takes as many steps. Summarising, $3\log\log n$ steps are sufficient for the load estimation sub-phase.

During the assignment sub-phase we play

$$\frac{\log\log n}{\log b} + \alpha + 1$$

collision games with parameters $a = 4(\alpha + 2)$, $b = a/2$, and $c = 1$, each of which (by Lemma 2.2) w.h.p. needs

$$\frac{\log\log n}{\log b} + 2$$

rounds. Additionally, passing of messages up and down the query trees takes at most

$$\frac{\log\log n}{\log b} + \alpha + 1$$

steps per round of the assignment sub-phase. Altogether, the assignment sub-phase needs a running time of

$$\left(\frac{\log\log n}{\log b} + \alpha + 1\right) \cdot \left(\frac{\log\log n}{\log b} + 2 \ + \ \frac{\log\log n}{\log b} + \alpha + 1\right) \ \leq \ (\log\log n)^2$$

steps. Hence, the complete phase needs at most

$$3\log\log n + (\log\log n)^2 \leq 2(\log\log n)^2$$

steps, which finishes the proof of the lemma. □

5.2.3 From Phase To Phase

In this section we show that if the system is λ-balanced for some phase, then with high probability, it also will be λ-balanced for the next phase, and, moreover, for the next polynomial number of phases.

Lemma 5.8 (Phase to phase)
Let $\alpha \geq 1$ be an arbitrary constant, let $\lambda = 20 \cdot (8\alpha + 16)^{2\alpha+5}$.

If the system is λ-balanced for some phase, then with a probability of at least $1 - 1/n^{\alpha}$, it remains λ-balanced for the next phase.

Proof. Fix some phase Π_i and assume the system to be λ-balanced for this phase, meaning that no server has a load exceeding

$$\lambda m_{\tau_i} = \lambda \left(\frac{M_{\tau_i}}{n} + 6\Delta T \right)$$

with M_{τ_i} being the complete system load at the first time step τ_i of phase Π_i. Assuming that each heavy server has found a balancing partner (Lemma 5.6), we now have to show that the load of any server is at most $\lambda m_{\tau_{i+1}}$ at the first step τ_{i+1} of the next phase Π_{i+1}.

We first investigate how the load of light, heavy, and neutral servers can change during a phase of $T = 2(\log\log n)^2$ steps, find a common upper bound on the load of any server at the end of the phase, and then show that this value is at most $\lambda m_{\tau_{i+1}}$.

Heavy servers. First consider a server P that has classified itself as heavy in phase Π_i. By definition of λ-balancedness, its load is at most λm_{τ_i} at the beginning of phase Π_i. By Lemma 5.6, it finds a light balancing partner and transfers

$$\frac{3}{\epsilon} \cdot \bar{\ell}_P \geq \frac{3}{\epsilon} \cdot \frac{2}{3} m_{\tau_i} = \frac{2}{\epsilon} m_{\tau_i}$$

tasks to it. Since P can have increased its load by at most ΔT tasks during phase Π_i, this leaves it with at most

$$\lambda m_{\tau_i} - \frac{2}{\epsilon} m_{\tau_i} + \Delta T = \left(\lambda - \frac{2}{\epsilon} \right) m_{\tau_i} + \Delta T$$

tasks at the end of the phase. Since $\lambda = 20(8\alpha + 16)^{2\alpha+5}$, $a = 4(\alpha + 2) = 4\alpha + 8$, $b = a/2$, $a - b + 1 = a - a/2 + 1 = 2\alpha + 5$, and $\epsilon = 1/(2(2a)^{a-b+1})$, we can conclude that

$$\lambda = 20(8\alpha + 16)^{2\alpha+5} = 20(2a)^{a-b+1} = 10/\epsilon,$$

and, hence, that P ends this phase with at most

$$\left(\lambda - \frac{2}{\epsilon} \right) m_{\tau_i} + \Delta T = \left(\frac{10}{\epsilon} - \frac{2}{\epsilon} \right) m_{\tau_i} + \Delta T = \frac{8}{\epsilon} m_{\tau_i} + \Delta T$$

tasks.

Light servers. A light server P can have a load of at most

$$\ell_P \leq 3 \cdot \bar{\ell}_P \leq 3 \cdot \frac{4}{3} m_{\tau_i} = 4 m_{\tau_i}$$

tasks at the first time step τ_i of phase Π_i. Some heavy server Q can transfer at most

$$\frac{3}{\epsilon} \cdot \bar{\ell}_Q \leq \frac{3}{\epsilon} \cdot \frac{4}{3} m_{\tau_i} = \frac{4}{\epsilon} m_{\tau_i}$$

tasks during a balancing action at the end of the phase to P, and P can increase its load by at most ΔT during the phase. This leaves P with at most

$$4 m_{\tau_i} + \frac{4}{\epsilon} m_{\tau_i} + \Delta T = \left(4 + \frac{4}{\epsilon}\right) m_{\tau_i} + \Delta T \leq \frac{5}{\epsilon} m_{\tau_i} + \Delta T$$

tasks at the end of phase Π_i (note that $4 < 1/\epsilon$).

Neutral servers. A neutral server P has a load ℓ_P of at most

$$\ell_P \leq \frac{6}{\epsilon} \cdot \bar{\ell}_P \leq \frac{6}{\epsilon} \cdot \frac{4}{3} m_{\tau_i} = \frac{8}{\epsilon} m_{\tau_i}$$

at time step τ_i. Since it also can have raised its load by at most ΔT, this leaves P with at most

$$\frac{8}{\epsilon} m_{\tau_i} + \Delta T$$

tasks at the end of phase Π_i.

Summing up. Now we have placed a bound of at most $(8/\epsilon) m_{\tau_i} + \Delta T$ on the load of any server at the end of phase Π_i. We now have to show that the system is λ-balanced for phase Π_{i+1}, i.e., that $\frac{8}{\epsilon} m_{\tau_i} + \Delta T \leq \lambda m_{\tau_{i+1}}$. Since obviously $m_{\tau_{i+1}} \geq m_{\tau_i} - \Delta T$, it suffices to show that $\frac{8}{\epsilon} m_{\tau_i} + \Delta T \leq \lambda (m_{\tau_i} - \Delta T)$. Using $\lambda = 10/\epsilon$ from above,

$$\begin{aligned}
& \tfrac{8}{\epsilon} m_{\tau_i} + \Delta T &\leq\ & \lambda(m_{\tau_i} - \Delta T) = \tfrac{10}{\epsilon}(m_{\tau_i} - \Delta T) = \tfrac{10}{\epsilon} m_{\tau_i} - \tfrac{10}{\epsilon} \Delta T \\
\Leftrightarrow\ & \left(1 + \tfrac{10}{\epsilon}\right) \Delta T &\leq\ & \tfrac{2}{\epsilon} m_{\tau_i} \\
\Leftrightarrow\ & m_{\tau_i} &\geq\ & \tfrac{\epsilon}{2}\left(1 + \tfrac{10}{\epsilon}\right) \Delta T = \left(\tfrac{\epsilon}{2} + 5\right) \Delta T
\end{aligned}$$

This last inequality is fulfilled since

$$m_{\tau_i} = \frac{M_{\tau_i}}{n} + 6\Delta T \geq 6\Delta T$$

and $\epsilon < 1$. This finishes the proof of the lemma. □

We are now ready to prove Theorem 1.6. Assume that the system is λ-balanced for some phase Π_i. We want to show that for any constant $k \geq 1$, with a probability of at least $1 - 1/n^\beta$ for some arbitrary constant $\beta \geq 1$, the system remains λ-balanced for at least n^k phases, that is, for phases $\Pi_{i+1}, \ldots, \Pi_{i+n^k}$. So we need to show that the probability that there exists a $j \in \{i+1, \ldots, i+n^k\}$ such that the system was λ-balanced in phases $\Pi_i, \ldots, \Pi_{j-1}$ but is not λ-balanced in phase Π_j can be upper bounded appropriately.

We know from Lemma 5.8, that if the system is λ-balanced for some phase, then it remains so for the next phase with a probability of at least $1-1/n^\alpha$ for some arbitrary constant $\alpha \geq 1$.

Hence, the probability that there is some phase $\Pi_{i+1}, \ldots, \Pi_{i+n^k}$ where the system arrives in a non-λ-balanced phase from a λ-balanced phase, can upper bounded by

$$n^k \cdot \left(\frac{1}{n}\right)^\alpha = n^{k-\alpha}.$$

If we now want to bound the overall failure probability by $1/n^\beta$,

$$n^{k-\alpha} \leq n^{-\beta} \;\Leftrightarrow\; k - \alpha \leq -\beta \;\Leftrightarrow\; \alpha \geq k + \beta$$

has to hold. Since all the lemmas of this chapter allow for an arbitrary α, letting $\alpha = k + \beta$ suffices to finish the proof of Theorem 1.6.

5.2.4 Expected Behaviour

In this section we prove Theorem 1.7, stating that the expected number of requests issued by any server during a phase is constant.

Recall that for any heavy server P, the development of its query tree is completely stopped as soon as a light partner has been found for P. Now call the query tree of a heavy server i-*active*, if its tree has a depth of at least i — this means that at least all the nodes on levels 1 to $i-1$ of its query tree represent either non-light or light but non-available servers. By Lemma 5.4, the probability that the server represented by some fixed node is light and still available, is at least $3/8$. Hence,

$$\text{prob}\,(\text{query tree is i-active}) \leq \left(\frac{5}{8}\right)^{b^{i-1}} .$$

Now developing the i-th level of the query tree causes b^{i-1} requests to be issued. Let the random variable X denote the number of requests issued to develop the query tree of some heavy server. Since a query tree has a depth of at most $\frac{\log\log n}{\log b} + \alpha + 1$, we have that

$$E[X] \leq \sum_{i=1}^{\frac{\log\log n}{\log b}+\alpha+1} b^{i-1} \cdot \text{prob}\,(\text{i-active}) \leq \sum_{i=1}^{\log\log n} b^{i-1} \cdot \left(\frac{5}{8}\right)^{b^{i-1}} = \mathcal{O}\,(1)\,.$$

This finished the proof of Theorem 1.7.

5.2.5 Worst Case Recovery

So far, we have shown that if the algorithm is λ-balanced for some phase, then with high probability, it will remain so for any polynomial number of phases. But of course, if we run the algorithm for a sufficiently long time, there can and will arise situations where the algorithm will no longer be able to distribute the load according to the bounds we have presented in this chapter.

A way to overcome this problem is to run a special recovery phase every now and then (every polynomial number of standard phases). In such a recovery phase, the system can be brought back into a typical state, that is, a state in which the condition of λ-balancedness will hold again, and therefore the standard algorithm will be able to keep on working for another polynomial number of phases. The important thing is that this must be possible from any arbitrary state, i.e., under the assumption of an arbitrary load distribution among the servers, and an arbitrary complete system load (even one server having a load of $\exp(\exp(\exp(\exp n)))$ or any other large number).

We are now going to describe a possible implementation of such a recovery phase. It is split into two sub-phases. The first sub-phase will reduce the maximum load of any server to the allowed bound of $\lambda(M/n + 6\Delta T)$ plus $\mathcal{O}(\log^3 n)$. In a second sub-phase, this remaining overload then also is

eliminated and we again have a well-behaved system, where no server has a load of more than some constant times the average. In the following, we are going to describe the two recovery sub-phases in some more detail.

The First Recovery Sub-Phase.

First we "freeze" the servers' load, that is, tasks generated during this first recovery sub-phase will at first not be considered (we will handle this additional load during the second recovery sub-phase; right now it suffices to imagine that these new tasks are put into some special queue).

This first sub-phase consists of $\alpha \log n$ *blocks* of $\beta \log n$ *rounds* each, for positive constants α and β. A round works as follows.

1. Play a constant number of $(n, \epsilon, a, b, c = 1)$ collision game with appropriate parameters, one after the other. In each game, a constant fraction of all servers issues request, such that afterwards, every server has issued one request (just deterministically divide the servers into groups of appropriate size). Since $c = 1$, after each such game, every server has accepted at most one query.

2. Fix one of these collision games. Every server P having issued a request during this game now inspects the load of its b successors. If there is one with at most half its own load, then these two equalise their load. (If there are more than one, P picks an arbitrary one of them.) If Q is this other server, and if ℓ_P and $\ell_Q \leq \ell_P/2$ denote the load of P and Q, respectively, then after this round, both have a load of at most
$$\frac{\ell_P + \ell_Q}{2} \leq \frac{\ell_P + \ell_P/2}{2} = \frac{3}{4}\ell_P$$
after the equalisation, i.e., the load of P was reduced by a factor of $3/4$, and the load of Q does not exceed P's load.

Let M denote the complete system load at the first time step of the recovery phase, and let L denote the maximum load of any server at the first time step of the recovery phase. The next lemma shows the following. The maximum load of any server is decreased by a factor of $3/4$ after the performance of a block as described above, as long as the maximum load of

any server is at least $4M/n$. Note that for a suitably chosen constant α, we then have after the complete first recovery sub-phase,

$$L \cdot \left(\frac{3}{4}\right)^{\alpha \log n} \leq M \cdot \left(\frac{3}{4}\right)^{\alpha \log n} \leq 4\frac{M}{n} \leq (8/\epsilon)\frac{M}{n}.$$

Lemma 5.9 (First recovery sub-phase)
Let $1 \leq i \leq \alpha \log n$ for some constant α, let $L^{(i-1)} \geq 4M/n$ denote the maximum load of any server at the beginning of the i-th of $\alpha \log n$ blocks. After the i-th block of the above protocol, we will have $L^{(i)} \leq L \cdot (3/4)^i$, w.h.p.

Proof. For every round in the i-th block, we call a server P *heavy*, if for its load ℓ_P,

$$L^{(i)} < \ell_P \leq L^{(i-1)}.$$

For every round in the i-th block, we call a server *light*, if its load is small enough so that the server can serve as a balancing partner for one of the heavy server of block i, i.e., the load of a light server is at most half the load of the least loaded heavy server.

The load of a heavy server is at least $4M/n$. Hence, for a light server Q that can balance with *any* heavy server we have

$$\ell_Q \leq 2M/n.$$

This applies for any round and at the beginning of any collision game of the i-th block. By a simple pigeonhole argument, at least half of all the servers fall into this category in every round and also at the beginning of every collision game. The servers neither heavy nor light are called *neutral*.

The proof of the lemma will be done by induction. At the beginning we have, due to our definition, $L^{(0)} \leq L \cdot (3/4)^0$. Now fix $1 < i < \alpha \log n$ and assume that $L^{(i-1)} \leq L \cdot (3/4)^{i-1}$.

Of course, during the performance of one block, the classification of any fixed server can change from round to round. For example, at the beginning of a block, server P can be heavy. It can be successful in finding balancing partners such that it becomes light during the block. At the end of the

block, it can be used as a balancing partner of another heavy server that has not yet found a balancing partner. In the following, we show that the load of a heavy server w.h.p. drops below the $L^{(i)}$ bound during the block, and that the load of a heavy and light server can not become larger than $L^{(i)}$ due to balancing actions.

Heavy servers. To show that no heavy server has a load larger than $L^{(i)}$ after the i-th block, we show that w.h.p. every heavy server finds a light one during the block.

A block consists of $\beta \log n$ rounds of the protocol presented above. With a probability of at most $(1/2)^b$, none of the b successors of P determined by the collision game during one round, is light, i. e. has a load of at most twice the average. Moreover, with a probability of at most

$$\left(\frac{1}{2}\right)^{b \cdot \beta \log n} = \left(\frac{1}{n}\right)^{b \cdot \beta},$$

during *all* the $\beta \log n$ rounds of this block, P is not successful in finding at least one server with a load of at most twice the average. We can conclude that with high probability, during a block consisting of $\beta \log n$ rounds, every heavy server with a load of at least four times the average finds at least one server with a load of at most twice the average among its b successors of the corresponding round. Hence, w.h.p. the load of every heavy server is decreased by a factor of $3/4$ at the end of the block.

Neutral servers. Neutral servers can not be used as balancing partners for heavy ones. Of course, balancing actions between light servers and neutral ones, or between two neutral servers, can not increase their load such that it is larger than $L^{(i)}$. Note that the load of a neutral server is at most $L^{(i)}$.

Light servers. Similar to the neutral servers, the load of a light server that is not used as a balancing partner for a heavy one is smaller than $L^{(i)}$ at the end of the i-th block. Since the load of a light server and the load of a heavy server (that uses the light one as balancing partner) will be equalised during a load transfer, there can only be a light server with

a load larger than $L^{(i)}$ at the end of the i-th block if there also is a heavy one with such a load. As this does not happen, w.h.p., the load of every light server is smaller than $L^{(i)}$ at the end of the i-th block. Recall that we do not consider tasks generated during this first recovery sub-phase, so newly generated or serviced tasks have no influence on the classification. Note that the load of a light server is at most $L^{(i-1)}/2 \leq L^{(i)}$.

□

The Second Recovery Sub-Phase.

So far, we haven't considered the load which is generated during the first recovery sub-phase. Its running time is $\mathcal{O}(\log^2(n) \cdot \log\log n)$; $\mathcal{O}\log^2 n$ blocks, each with a constant number of constant number of collision games with running time $\mathcal{O}(\log\log n)$. Since a server can generate up to Δ tasks per step, this means that now the maximum load of any server can be bounded by

$$\begin{aligned} &\frac{8}{\epsilon}\frac{M}{n} + \mathcal{O}(\log^3 n) \\ &\leq \frac{8}{\epsilon}\left(\frac{M}{n} + 6\Delta T\right) + \mathcal{O}\left(\log^3 n\right) \\ &= \frac{8}{\epsilon}\left(\frac{M}{n} + 6\Delta T\right) + L, \end{aligned}$$

where M is the current complete system load, taken at the first time step of the second recovery sub-phase. Hence, it remains to eliminate the additive term of $L = \mathcal{O}(\log^3 n)$ to obtain a λ-balanced system.

This second recovery sub-phase consists of $\alpha \log\log n$ blocks for some constant α. Each block is similar to a phase of the standard algorithm. First comes a load estimation, where the servers have to play an (a, b, c) collision game (each server issues a query). This creates a random graph with out-degree b and in-degree at most c. Each server calculates the load of its T_P, which now has to have $\Theta(\log^8 n)$ nodes. This means that the T_P's have to be a little deeper, but just by some constant factor (more on this later).

Similar to a standard phase, we try to find balancing partners for the heavy servers. Let $\bar{\ell}_P$ denote the estimated average load of server P. Servers P

with a load of at least $(4.5/\epsilon)\bar{\ell}_P$ are classified as heavy (ϵ is the constant of the collision game from Lemma 5.5, and just the value which has been used to classify during the standard algorithm). We build query trees using collision games with $c = 1$, but now we do not have a fixed classification for light servers, but proceed a little bit different. Assume that along with the queries, the load of the heavy server is passed, on whose behalf the query tree is developed. Then each server with a load of at most half the one of the heavy server "on the fly" classifies itself as being a suitable balancing partner and sends an offer message. Like in the first recovery sub-phase, a server P that has found a balancing partner Q, equalises its load with Q. Again, after the equalisation, the load of P was reduced by a factor of at least $3/4$, and the load of Q does not exceed P's load.

Similar to Lemma 5.9, we want to show that the overhead L decreases by a factor of at least $3/4$ during every block. This implies that after $\alpha \log\log n$ blocks, the overhead is "eaten up", for a suitable chosen constant α.

Let $m_i = M_i/n + 6\Delta T$, where M_i denotes the complete system load at the beginning of the i-th block of the second recovery sub-phase. At first we bound the quality of the load estimation. This can be done similar to Lemma 5.3. We now have a larger bound on the load of any server to start with (the additional term of $O(\log^3 n)$ compared to the load difference of Lemma 5.3), and this is why we now need larger trees T_P. But it can easily be shown that again for every server P we have for its load estimation $\bar{\ell}_P$,

$$\frac{2}{3} \cdot m_i \leq \bar{\ell}_P \frac{4}{3} \cdot m_i,$$

with high probability.

Lemma 5.10 (Second recovery sub-phase)
Let M_i be the complete system load at the beginning of the i-th block, $0 < i \leq \alpha \log n$. Let $L^{(i)} = (3/4)^i L$, and let $m_i = M_i/n + 6\Delta T$. Let $(8/\epsilon)m_i + L^{(i-1)}$ denote the maximum load of any server at the beginning of i-th block, $i \geq 1$.
After the i-th block of the above protocol w.h.p. we will have a maximum load of at most

$$\frac{8}{\epsilon} m_{i+1} + L^{(i)} \leq \frac{8}{\epsilon} m_{i+1} + L \cdot \left(\frac{3}{4}\right)^i.$$

Proof. We call a a server *relevant* in the i-th block if its load is larger than $(6/\epsilon)m_i$. A server P with a load of at least $(4.5/\epsilon)\bar{\ell}_P$ classifies itself as heavy. For the load of a non-heavy server Q it has to hold

$$\frac{4.5}{\epsilon} \cdot \bar{\ell}_Q \leq \frac{4.5}{\epsilon} \cdot \frac{4}{3} m_i = \frac{6}{\epsilon} m_i,$$

Hence, every relevant server P will classify itself as heavy, w.h.p. Hence, it will try to find another server Q in order to equalise its load with Q.

Again, we call a server *light* in the i-th block, if its load is small enough so that the server can serve as a balancing partner for a heavy server of phase i (half the load of the lightest heavy server). A server P that is classified as heavy has a load of at least

$$\frac{4.5}{\epsilon} \cdot \bar{\ell}_P \geq \frac{4.5}{\epsilon} \cdot \frac{2}{3} m_i = \frac{3}{\epsilon} m_i.$$

Hence, all servers with a load of at most $(1.5/\epsilon)m_i$ can play the role of balancing partners for them (at least $3/4$ of all servers). The remaining servers are called neutral. Note that the load of both the neutral and the light servers is at most $6/\epsilon m_i$

Again, the proof of the lemma will be done by induction. At the beginning, we have that the maximum load of any server is bounded by

$$\frac{8}{\epsilon} m_1 + L^{(0)} = \frac{8}{\epsilon} m_1 + L \cdot (3/4)^0 = \frac{8}{\epsilon} m_1 + L.$$

Now fix $1 < i < \alpha \log\log n$, and assume that at the beginning of the i-th block, the maximum load is bounded by

$$\frac{8}{\epsilon} m_i + L^{(i-1)} = \frac{8}{\epsilon} m_i + L \cdot (3/4)^{i-1}.$$

In the following, we show that w.h.p., the load of a heavy server drops below the $(8/\epsilon)m_{i+1} + L^{(i)}$ bound by the end of the block, and that neither for heavy nor for light servers their load can not become larger than $(8/\epsilon)m_{i+1} + L^{(i)}$ due to balancing actions.

Relevant servers. At first we will show that w.h.p. each relevant server will find a server that is suitable to equalise its load with. Each relevant server has a real load of at least

$$\frac{4.5}{\epsilon} \bar{\ell}_P \geq \frac{4.5}{\epsilon} \cdot \frac{2}{3} m_i = \frac{3}{\epsilon} m_i.$$

Hence, there are at most $(\epsilon/3)n$ heavy servers. All servers with a load of at most $(1.5/\epsilon)m_i$ can play the role of balancing partners any relevant one. This means that at least 3/4 of all the servers are suitable as partner. Similar to Lemma 5.4, we can show that the probability is at least 3/8 that a fixed node of the query tree is light. Similar to Lemmas 5.5 and 5.6, we can show that w.h.p. each of the collision games needed to construct the query trees succeed in resolving the requests, and that w.h.p. every relevant server gets assigned a suitable server in order to balance the load. Since only the load estimation has a slightly larger running time ($9 \log\log n$ instead if $3 \log\log n$) the whole running time of a block can still be upper bounded by $T = 2(\log\log n)^2$.

Due to the induction hypothesis, a relevant server P has a load of at most $(8/\epsilon)m_i + L^{(i-1)}$ at the beginning of the block. After balancing the load with a suitable server, they both have a load of at most

$$\frac{\left(\frac{8}{\epsilon}m_i + L^{(i-1)}\right) + \frac{1}{2}\left(\frac{8}{\epsilon}m_i + L^{(i-1)}\right)}{2} \leq \frac{6}{\epsilon}m_i + \frac{3}{4}L^{(i-1)} = \frac{6}{\epsilon}m_i + L^{(i)}.$$

During block i, P can generate up to Δ new tasks per step, and the other server can consume up to Δ tasks per step. Similar to Lemma 5.8, we can now show that this load is at most $(8/\epsilon)m_{i+1} + L^{(i)}$, compared to the average load at the beginning of block $i+1$.

Neutral servers. Neutral servers can not be used as balancing partners for heavy ones. Due to the definition of relevant and light, a neutral server has a load of at most

$$\frac{4.5}{\epsilon}\bar{\ell}_P \leq \frac{4.5}{\epsilon} \cdot \frac{4}{3}m_i = \frac{6}{\epsilon}m_i.$$

Again we can show that this load is at most $(8/\epsilon)m_{i+1} \leq (8/\epsilon)m_{i+1} + L^{(i)}$, compared to the load at the beginning of block $i+1$.

Light servers. Similar to the neutral servers, a light server that is not used as balancing partner has a load of at most $(8/\epsilon)m_{i+1} + L^{(i)}$ after the i-th block.

The fact that also the load of light servers *being* used as balancing partners is at most $(8/\epsilon)m_{i+1} + L^{(i)}$, follows from the bound for relevant servers, as

these servers do not have a larger bound than the relevant servers being their balancing partner. □

CHAPTER 6

Simulations

In this section we present the results of some simulations we performed based on the algorithms introduced in the previous sections. Note that in the remainder of this section all logarithms are base 2, and that when taking the logarithm of some number we always take the ceiling of said logarithm in order to obtain integer numbers (so, when calculating $\log\log n$ we actually calculate $\lceil \log_2(\lceil \log_2 n \rceil) \rceil$.

SECTION 6.1

Technical Difficulties

First, we would like to point out some (well known) technical problems inherent to the type of simulations we have performed.

Size of integers. Assume that the output of a simulation can be expressed as a function $f(n)$, where n is the input size (like the number of servers in our case). Now it is not easy to decide whether $f(n)$ is *constant* or if $f(n)$ *grows with* n, if the size of the integers which are used within the simulation is limited. If the machine which the simulation runs on has, say, 32-bit integers then the maximum representable integer number is $2^{32} - 1$. If now the simulator solely makes use of standard integer based arithmetics then the double-log of this maximum integer is $\log\log(2^{32} - 1) < 5$. Obviously, in such a case one cannot easily decide whether $f(n)$ is constant,

double-logarithmic in n, or even logarithmic (since the logarithm of this number is less than 32). Unfortunately, even a 64-bit architecture doesn't make things much easier.

An obvious way to overcome this problem is to rely to some way of using larger (or even "arbitrarily" large) integers and base the corresponding parts of the simulation on them instead on the standard integers. Although this seems to be an elegant way out of the trouble, it doesn't come without a price, for this can *dramatically* increase the running time of the simulations. In general, the larger the maximum range of integers, the larger the running time will be.

Closely related and specific to the algorithms presented in this work is the problem of fixing certain numerical values. Consider, for instance, the phase length of the first algorithm presented for the stochastical generation model. Here, we have phase lengths of $T = (\log\log n)^2/10$. In order to obtain phases of "reasonable" length, say, $T \geq 10$, we would have to have

$$\begin{aligned}
\frac{(\log\log n)^2}{10} &\geq 10 \\
(\log\log n)^2 &\geq 100 \\
\log\log n &\geq 10 \\
\log n &\geq 2^{10} = 1024 \\
n &\geq 2^{1024},
\end{aligned}$$

which, for reasons of integer range as well as memory consumption and simulation running time, obviously is way out of the question. Even for just $T \geq 3$ we would have to have $n \geq 2^{32}$. For these reasons we need to find some sort of trade-off between practicability and closeness to the "true algorithm" which we hope to actually have found. Anyway, even if it now seems that the algorithms are as far from reality as possible, it turns out that after some "constant tweaking" (to obtain reasonable parameters like phase length, light/heavy thresholds and so on), they behave very nicely in simulation. Still it is not easy to apply them to real life problems on real existing parallel machines since there *is* a demand for a large number of servers (see above) and this demand is not easily fulfilled. See Section 7 for more on this subject.

Random Number Generation. All of our algorithms are randomised and the stochastical load generation model implies that the generation of tasks obeys some probability distribution. Therefore, during our simulations we need to access some source of randomness. Clearly, the quality of the random number generator can have significant influence on the quality of the results, especially if there is need for *many* random numbers, as is the case for our simulations.

We implemented the simulator in `ANSI C`, so there is always the choice of using the random number generator included in the standard `C` library. Although this one might be useful and sufficient for occasionally throwing a die or two, the standard seems not to be sufficient for our needs as it states that the random numbers have to come out of a range of 0 to at least $2^{15} - 1 = 32767$. Although most modern implementations allow for a much wider range, or even for the choice among at least two generators (one for a relatively small range and one for the complete integer range, which nowadays usually is 32 bit), there is nothing said about the statistical quality (like period length). Therefore, we made use of an external library which provides a wealth of both different high-quality random number generators as well as functions to put certain probabilistic distributions on top of them. For more details, see [BDG+99]. We chose to use the `MT19937` generator by Matsumoto and Nishimura (see [MN98]) which is a variant of the twisted generalised feedback shift-register algorithm, and is known as the *Mersenne Twister* generator. It has a Mersenne prime period of $2^{19937} - 1$ and is equi-distributed in 623 dimensions. It still is reasonably fast.

SECTION 6.2

Basic Simulation Rules

Each time we performed a simulation of one of the balancing algorithms, we made use of the following basic parameters. We set $n = 50,000$ (the number of servers), and simulated the system for at least $2,000,000$ time steps. For each set of parameters (varying probabilities in the case of the stochastical load generation model, or varying generation schemes in the case of the adversarial one), we always performed 10 runs in a row and

then calculated the average of the crucial resulting parameters (mostly the maximum load of any server). These average values then found their way into the plots shown below.

SECTION 6.3

The Collision Protocol

In this section we drop a few brief words about the performance of the collision game. In all the settings we make use of the collision game it can be shown that all the requests are resolved within $\mathcal{O}(\log\log n)$ rounds of the protocol, with high probability. Furthermore, it can be shown that expectedly the protocol finishes within just a constant number of rounds. We performed a couple of simulations within various settings (various parameter sets (n, ϵ, a, b, c) – always according to how they are set within the algorithms), and *all* of them showed that the analysis really is a worst-case analysis, since in *not a single case* the protocol needed more than four rounds to resolve the requests just perfectly. We omit figures here, since there is not much to learn from them.

SECTION 6.4

The Balancing Algorithm AlgStochMultiColl

Throughout this and the next section, we present some simulation results concerning the stochastical load generation model. Recall that in this model, at each time step each server generates a task with a probability of p and services a task (if there is any) with a probability of $p(1+\epsilon)$, where $\epsilon > 0$ is some constant. In the following two sections, we provide the simulation results for both of our algorithms for the stochastical load generation model in more detail.

This section refers to the algorithm AlgStochMultiColl presented in Section 3. The length of a phase here is $T = (\log\log n)^2/10$. With $n = 50,000$ we therefore have $T \approx 2$. Obviously, this is not very reasonable, since alone during the assignment sub-phase we play $\log\log n/\log b$

collision games, each with a running time of $\log\log n/\log b + 2$, where $b = 2(\alpha + 3)$ and α is the constant giving the probability bound. So, with $\alpha = 1$ we have $b = 8$ and $\log\log n/\log b \approx 2$, and we would have to play 2 collision games, each with running time 4 — which certainly does not fit well into a phase of length 2. Note that this is not a problem of the algorithm itself or its analysis; the problem is that the analysis holds for "n large enough", where this large enough n is not reasonably obtainable within the simulation.

This problem can be overcome if we just expand the single phases of our simulation such that all the necessary actions can be performed. Note that this approach leads to an upper bound on the performance of the true algorithm since during longer phases in the worst case servers can generate more tasks (and this worst case is what the analysis is based on). Further note that we do not change any other parameters like heavy/light thresholds or the number of tasks to be transferred in case of a balancing action. Since, by analysis, we have to play 2 collision games each of running time 4, we decided to "expand" the phases from 2 to 10 steps.

As mentioned before, we have run the simulation with $n = 50,000$ servers for $2,000,000$ time steps ($200,000$ phases each of length 10). We have a heavy threshold of $8T = 16$, a light threshold of $T = 2$, and during a balancing action we transfer $4T = 8$ tasks. We present three plots with different task generation probability, $p \in \{0.2, 0.5, 0.8\}$. In each case we set $\epsilon = 0.01$ (recall that task generation probability p implies service probability $p(1 + \epsilon)$). Each plot depicts the maximum load of any server over the running time (measured in phases), for the balancing system (the `sim1-bal-x.out` curve) as well as for a non-balancing system, with no balancing actions at all (the `sim1-nobal-x.out` curve), where x denotes the task generation probability. See Figures 6.1, 6.2, and 6.3 for the outcomes.

As can be seen, for all three generation probabilities, the curves for the balancing system are close to the x-axis whereas in the non-balancing system the curve is much higher. It is worth noting that the larger p becomes, the lower the maximum load is in the non-balancing system. This is can be explained by looking at what already is stated in the analysis, namely that ν_i, the probability for a single server to have load i, can be expressed as $(1 - 1/\mu) \cdot (1/\mu)^i$, where $\mu = \frac{\epsilon+(1-p(1+\epsilon)}{1-p(1+\epsilon)}$. Figure 6.4 shows the curves for ν_i, $0 \leq i \leq 100$, for $p \in \{0.2, 0.5, 0.8\}$ and $\epsilon = 0.01$. As can be seen, for

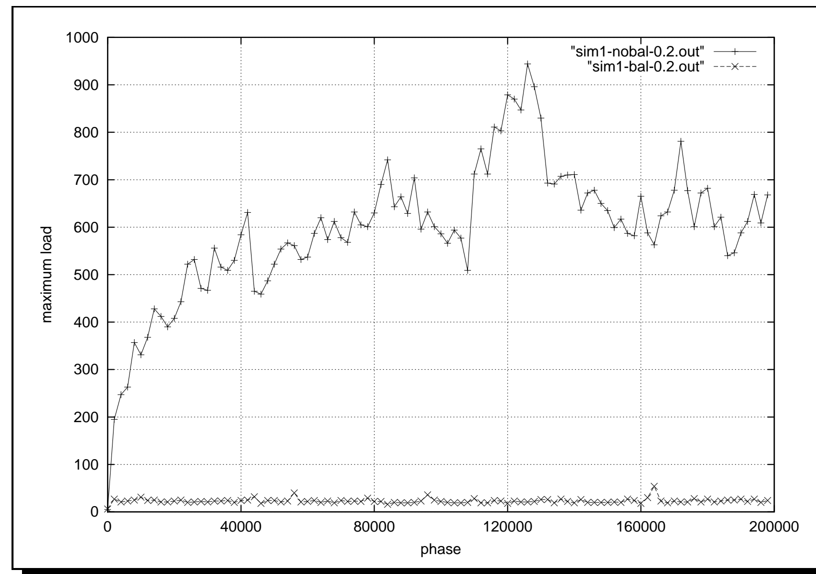

Figure 6.1: Stochastical generation, first algorithm AlgStochMulti-Coll, $p = 0.2$

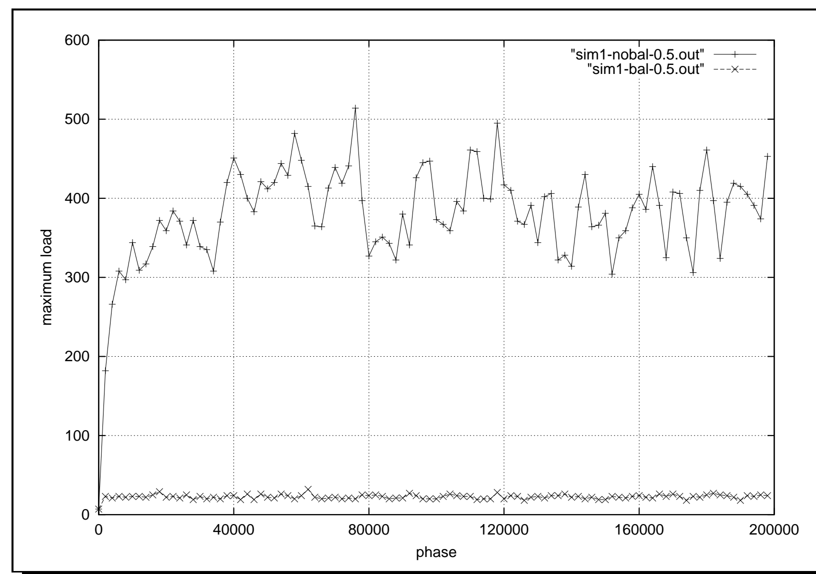

Figure 6.2: Stochastical generation, first algorithm AlgStochMulti-Coll, $p = 0.5$

$i \geq 37$, the curve for $p = 0.8$ is below the two other ones for 0.2 and 0.5, implying that for $p = 0.8$ the probability for a server to have "many" tasks is less than in the other two cases.

Additionally, one can calculate the expected load of a server in the non-balancing system. For $p = 0.2$ we have 79.8, for $p = 0.5$ we have 49.5, and for $p = 0.8$ we have 19.2.

Anyway, the plots clearly show that in the balancing system the maximum load is much lower than in the non-balancing one.

We furthermore performed "highly heuristic" simulations in order to see how the algorithm reacts on changing certain parameters. Recall that during the assignment sub-phase we play some number of $(a, b, 1)$ collision games, where a and b depend on α, the value giving the probability bound. In the original setting we played two collision games running for four rounds each, with parameters $a = 16$ and $b = 8$. Since a and b have a strong influence on the communication overhead of the algorithm (recall that a measures the number of queries issued for any request), it is quite interesting to see how the algorithm behaves if we scale those parameters down. We have run simulations where we played two collision games running for three rounds each, and we chose $a = 3$ and $b = 2$, and still the algorithm was able to bound the maximum load of any server. See Figures 6.5, 6.6, and 6.7 for the outcomes of the simulation with generation probabilities 0.2, 0.5, and 0.8, respectively.

As can be seen, although the analysis requires certain parameters, in the simulation the algorithm still works under relaxed conditions.

SECTION 6.5

The Balancing Algorithm AlgStochSingleColl

In this section we focus on our second algorithm AlgStochSingleColl (see Section 4 for details) for the stochastical load generation model. The load generation is just as described above, just that the algorithm has phases of length $T = \mathcal{O}(\log \log n)$ and reduces the maximum load of any server also to $\mathcal{O}(\log \log n)$ (which, due to constants in the $\mathcal{O}$ calculus,

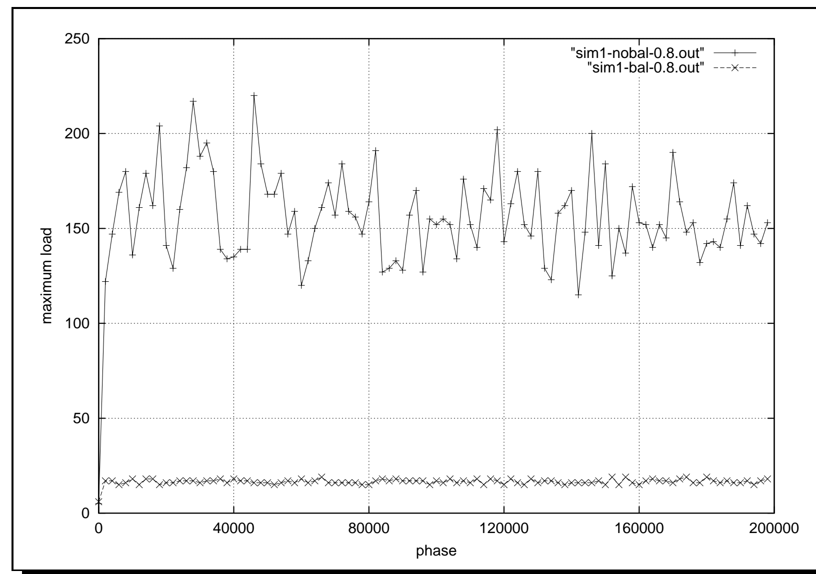

Figure 6.3: Stochastical generation, first algorithm AlgStochMulti-Coll, $p = 0.8$

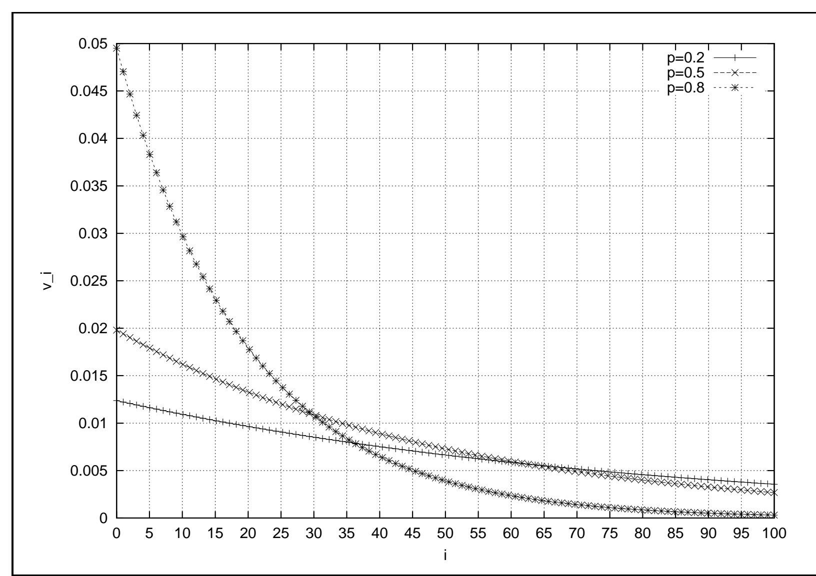

Figure 6.4: Comparison of ν_i for $p \in \{0.2, 0.5, 0.8\}$

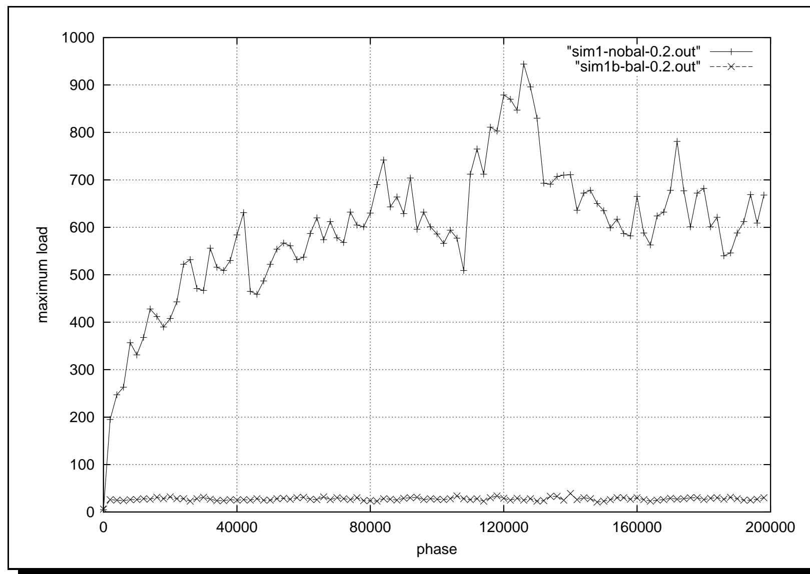

Figure 6.5: Stochastical generation, first algorithm ALGSTOCHMULTI-COLL, communication optimised, $p = 0.2$

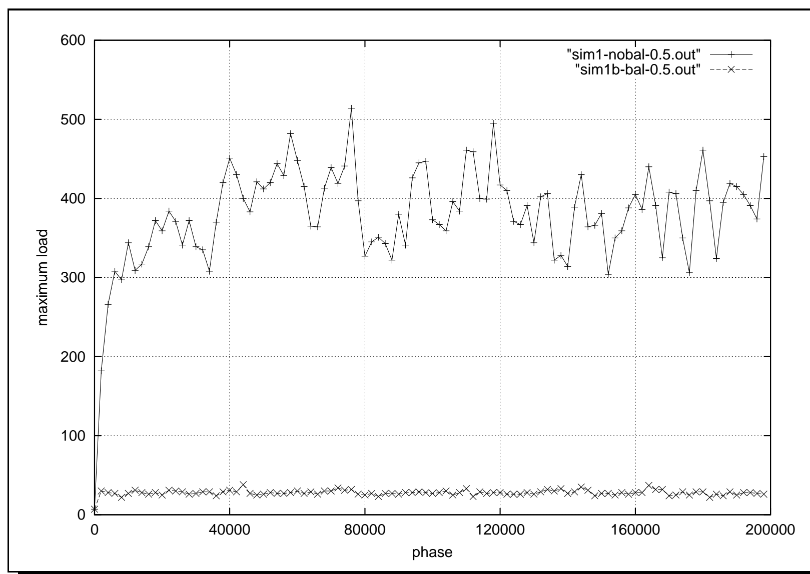

Figure 6.6: Stochastical generation, first algorithm ALGSTOCHMULTI-COLL, communication optimised, $p = 0.5$

unfortunately does not show up here in the simulations).

For $n = 50,000$ we have a phase length of $T = 3 \log\log n = 12$, so there is no need to "stretch" the phases artificially (as we had to do before). Figures 6.8, 6.9, and 6.10 show how the algorithm performs for varying generation probabilities (again, $p \in \{0.2, 0.5, 0.8\}$) and fixed $\epsilon = 0.01$. Again, there are two curves in each plot, one for the non-balancing system and one for the balancing one.

Note that we now have a heavy threshold of $8T = 96$, which explains why the curve for the balancing system is higher than it has been for our first algorithm. But this second algorithm has a quite notable property in that it behaves very nicely as far as scaling of constants is concerned (here we exclude the phase length but only mean the two thresholds (heavy/light) and the number of tasks to be transferred). The original algorithm uses values of $8T = 96$ for the heavy threshold, $T = 12$ for the light threshold, and $4T = 48$ as the number of to-be-transferred tasks (the numerical values refer to our choice of $n = 50,000$). The algorithm still is able obtain a stable system if we scale these constants by a factor of $1/8$, that is, with a heavy threshold of $T = 12$, a light threshold of $T/8 \approx 1$, and $T/2 = 6$ tasks to be transferred during each balancing action. For the outcomes of the simulation, refer to Figures 6.11, 6.12, and 6.13.

Furthermore, we tried to do the same as we did for the first algorithm, namely reducing the communication complexity by means of reducing the crucial parameters of the collision games. Recall that in this second algorithm, we no longer play one $(a, b, 1)$ collision game after the other, but play just one single collision game where all servers take part, and use the result as sort of an access graph where heavy servers try to find light ones within a certain distance from themselves. Within the original setting, we had $a = 7$, $b = 6$, and $c \approx 1000$, and said distance was five. We now tried to reduce these values such that the algorithm still is able to bound the maximum load of any server "reasonably", which turned out to work for values $a = 4$, $b = 3$, $c = 8$, and a search depth of 4. See Figures 6.14, 6.15, and 6.16 for the outcomes of the simulations for generation probabilities 0.2, 0.5, and 0.8, respectively. Note that this plot refers to the modified algorithm where we additionally down-scaled the light/heavy thresholds and the number of tasks to be transferred during balancing actions.

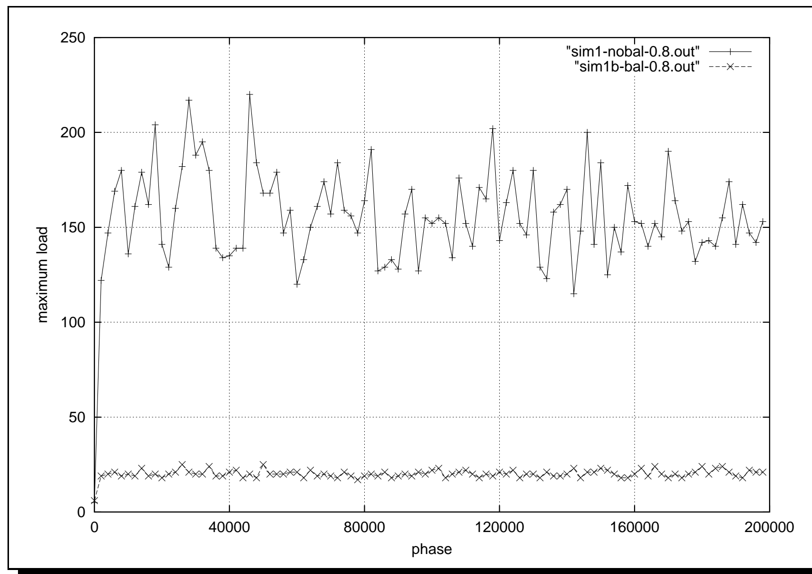

Figure 6.7: Stochastical generation, first algorithm ALGSTOCHMULTI-COLL, communication optimised, $p = 0.8$

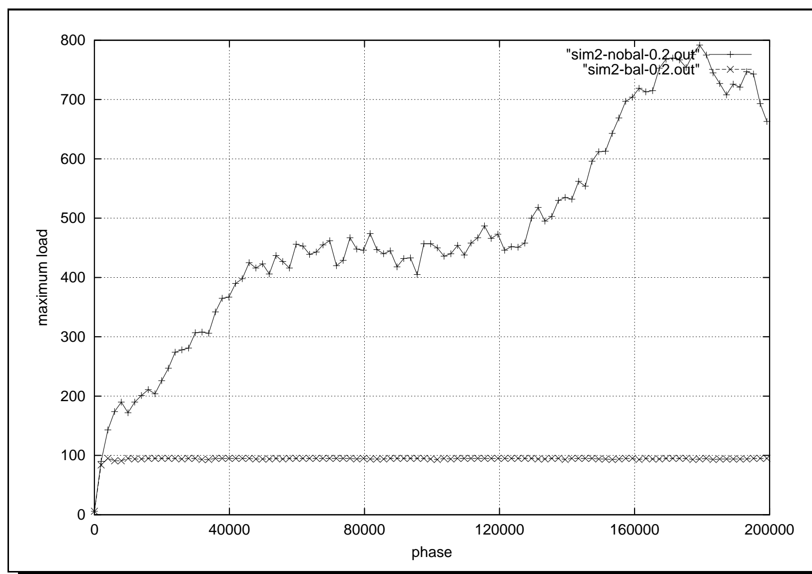

Figure 6.8: Stochastical generation, second algorithm ALGSTOCHSINGLE-COLL, $p = 0.2$

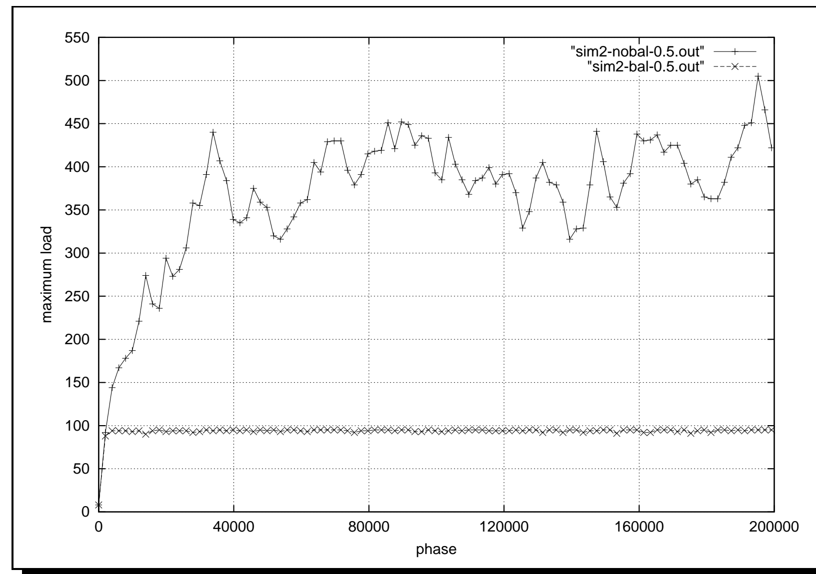

Figure 6.9: Stochastical generation, second algorithm ALGSTOCHSINGLE-COLL, $p = 0.5$

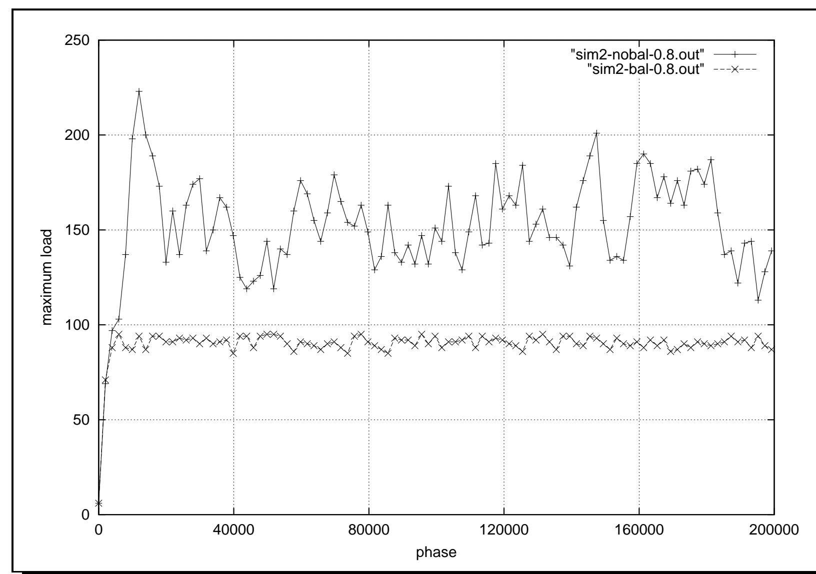

Figure 6.10: Stochastical generation, second algorithm ALGSTOCHSINGLE-COLL, $p = 0.8$

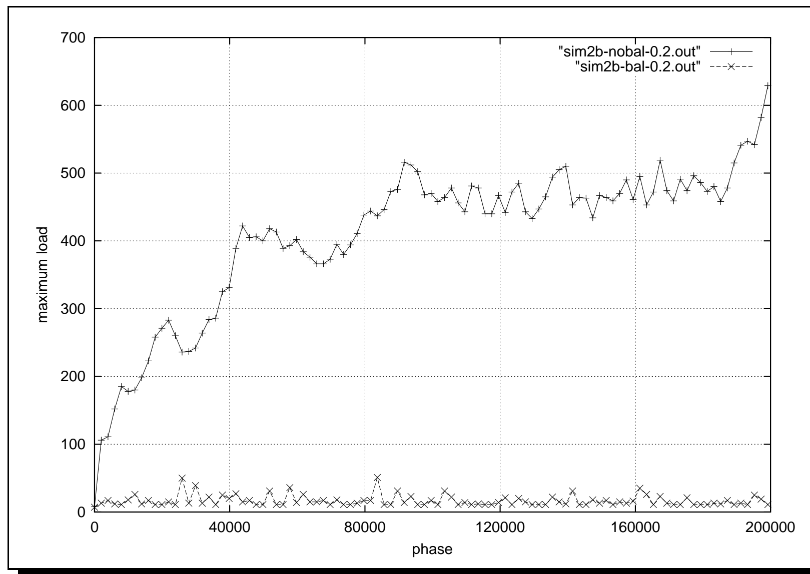

Figure 6.11: Stochastical generation, second algorithm AlgStochSingle-Coll, down-scaled constants, $p = 0.2$

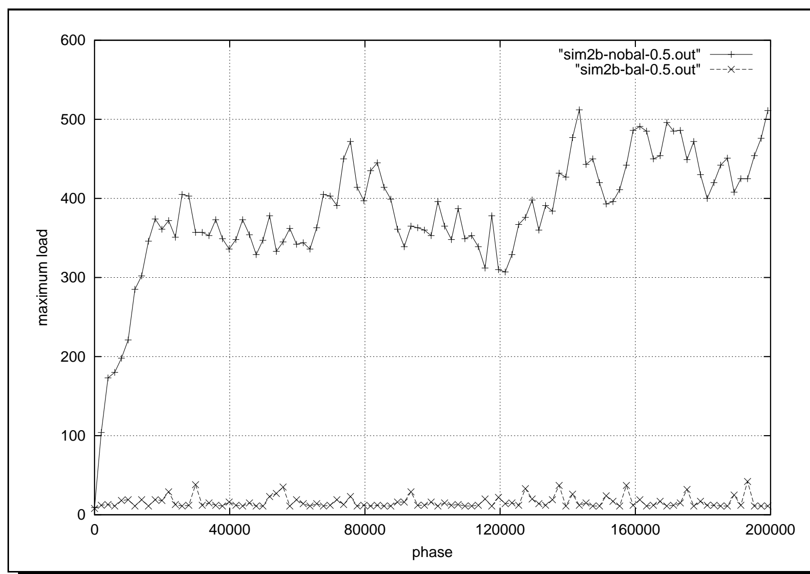

Figure 6.12: Stochastical generation, second algorithm AlgStochSingle-Coll, down-scaled constants, $p = 0.5$

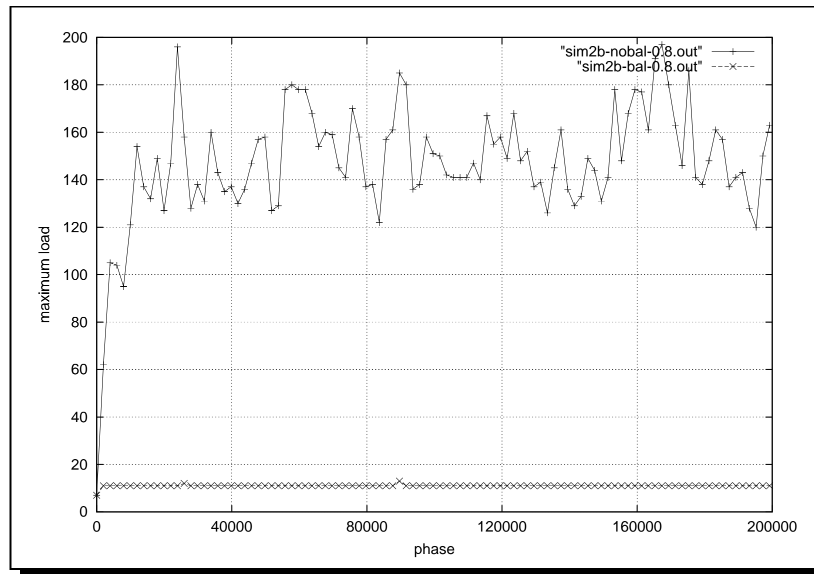

Figure 6.13: Stochastical generation, second algorithm AlgStochSingleColl, down-scaled constants, p = 0.8

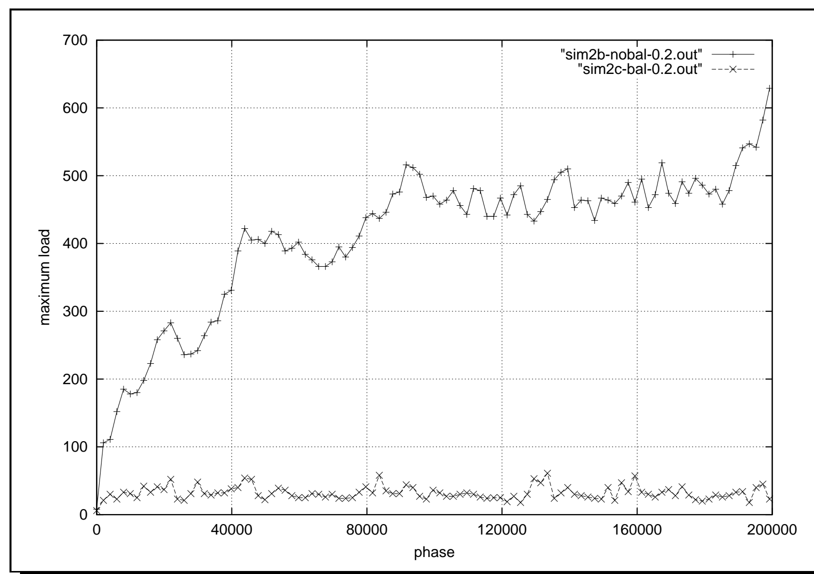

Figure 6.14: Stochastical generation, second algorithm AlgStochSingleColl, down-scaled constants, communication optimised, p = 0.2

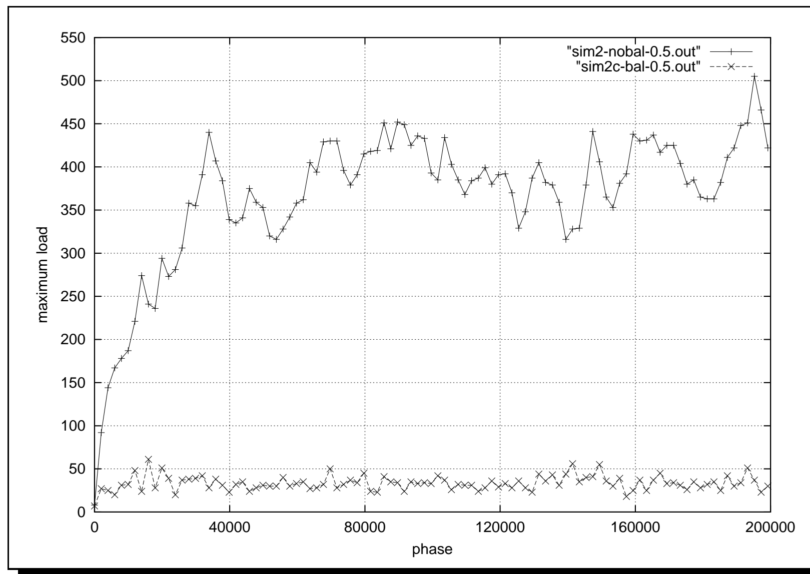

Figure 6.15: Stochastical generation, second algorithm ALGSTOCHSINGLE-COLL, down-scaled constants, communication optimised, $p = 0.5$

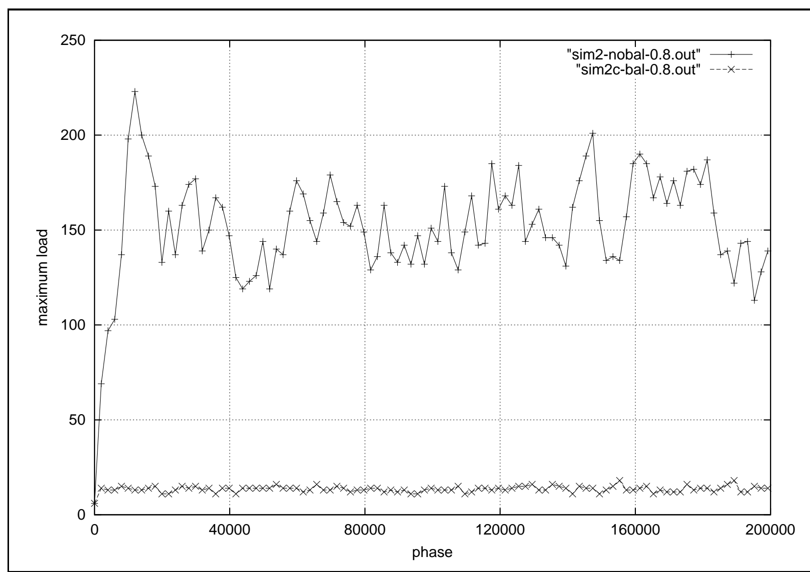

Figure 6.16: Stochastical generation, second algorithm ALGSTOCHSINGLE-COLL, down-scaled constants, communication optimised, $p = 0.8$

SECTION 6.6

The Balancing Algorithm ALGADV

In this section we present some simulation results for the adversarial generation model. Recall that in the basic version of this model we allow a server to change its load due to task generation and consumption by up to Δ tasks, where $\Delta \geq 1$ is some constant. Before we come to the actual simulations, let us point out two major problems concerning this algorithm.

First, there is again the problem of unreasonable numerical values. Although constants in the analysis, they can no longer be seen as constants regarding the simulation. For instance, consider the heavy threshold of $(6/\epsilon) \cdot \bar{\ell_P}$, where $\bar{\ell_P}$ is the system load estimation of server P, $\epsilon = 1/(2 \cdot (2a)^{a-b+1})$, $a = 4(\alpha + 2)$, $b = 2(\alpha + 2)$, and $\alpha \geq 1$. Since $\alpha \geq 1$ we have $a \geq 12$, and, therefore, $6/\epsilon \geq 6 \cdot 2 \cdot (2 \cdot 12)^7 > 2^{35}$ — clearly unreasonable. We will see that the algorithm itself does not require such enormous constants in simulations. Actually, it is rather well-behaved as far as such things are concerned.

Second, there is a problem somehow related to the one from above. Assume a scenario in which a $1/c$ fraction of the servers generate Δ tasks in each step and do not service any, whereas the remaining servers just service up to Δ tasks in each step (this models the well-known and often used "farmer" approach). Now assume that the system ran for t steps, and no load balancing at all took place (that is, the n/c generating servers just generate and never distribute their tasks anywhere such that no task was serviced so far). Clearly, at this point of time, the maximum load of any server is $t \cdot \Delta$, whereas the average system load is $((n/c) \cdot t \cdot \Delta)/n = t \cdot \Delta/c$. This yields a maximum/average load ratio of c. Thus, if c is a constant, we always have a constant factor between the maximum load of any server and the average load, *even without load balancing*. Recall that throughout the simulations we use $n = 50,000$ servers with $\log n \approx 16$. So, if we thought of, say, $100 \log n$ servers generating tasks, this would imply that even without load balancing we would not obtain a factor worse that 31 between the maximum load and the average load; if we would let every fourth server generate, this would result in a factor of 4.

Clearly, given these two problems, it neither makes sense to apply the

original thresholds of the analysis, nor to direct the interest to the factor between maximum load and average load only. Therefore, we scale down constants as well as investigate the behaviour of the algorithm with respect to another measure, namely the *stability* of the system. We investigate the development of the system load as time passes. If the total injection (the sum of all generated tasks) is less than the total service capacity, then one would expect from a "good" balancing algorithm that the system load remains bounded (does not grow to infinity over time), what clearly is *not* the case for a non-balancing system with this generation scheme. There, the system load would increase proportionally to the number of generating servers.

Recall that for the algorithm AlgAdv for the adversarial generation we divide time into phases of length $T = 2(\log\log n)^2$. With $n = 50,000$ we then have $T \approx 32$, which is sufficient to play all the collision games we need to play for this algorithm (during the load estimation and the assignment sub-phase, respectively). We changed the classification thresholds to $2T$ for the light threshold and $8T$ for the heavy threshold (recall the $\approx 2^{35}$ in the original analysis), and during balancing actions we transferred $4T$ tasks. We did not change the (a, b, c) parameters of the collision games, however.

We presented the algorithm with two different numbers of generating processors; $n/10$ and $n/100$, respectively, and in each case we simulated with $\Delta = 1$ and $\Delta = 10$. We have four figures 6.17 to 6.20, one for every number of generating servers in combination with every Δ. Each figure contains two curves (total system load over time), where one curve is for the balancing system and the other one for the non-balancing one. Note that the y-axis has a logarithmic scale (the curves for the non-balancing systems are linearly increasing).

For some conclusions concerning the simulations and a few further words comparing the simulations to the algorithm as presented in the analytical part the this work, refer to Section 7.

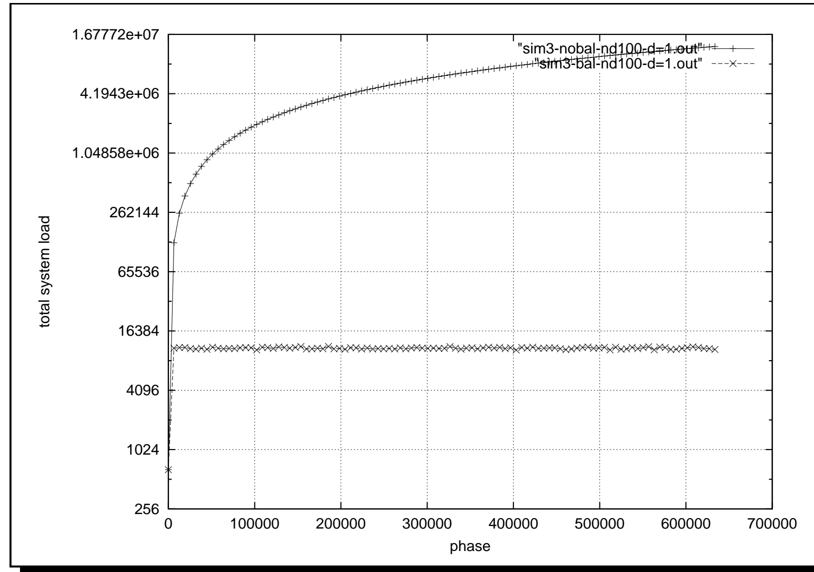

Figure 6.17: Adversarial generation, algorithm AlgAdv, $n/100$ generators, $\Delta = 1$ (logarithmic scale on y-axis)

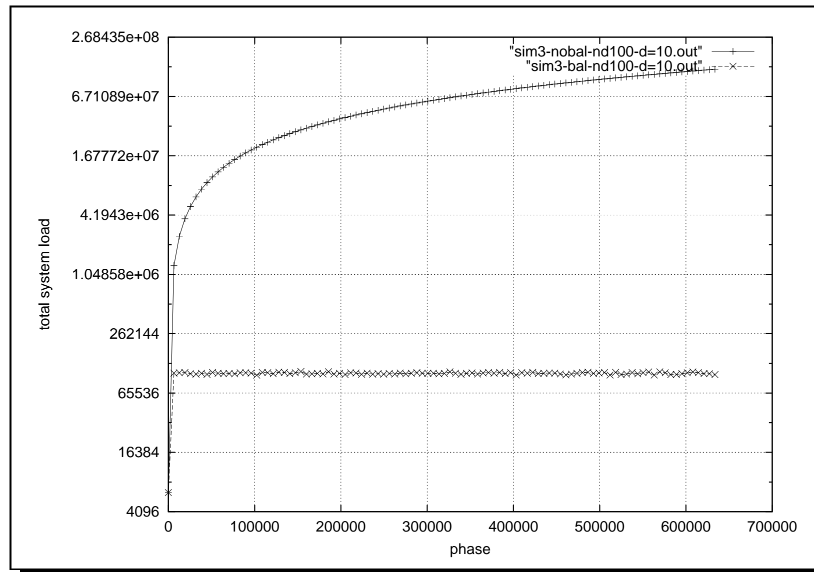

Figure 6.18: Adversarial generation, algorithm AlgAdv, $n/100$ generators, $\Delta = 10$ (logarithmic scale on y-axis)

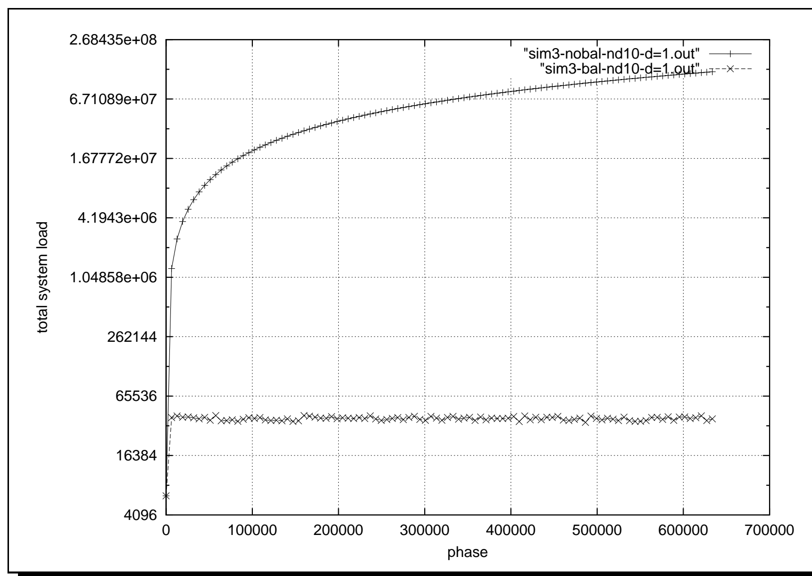

Figure 6.19: Adversarial generation, algorithm ALGADV, $n/10$ generators, $\Delta = 1$ (logarithmic scale on y-axis)

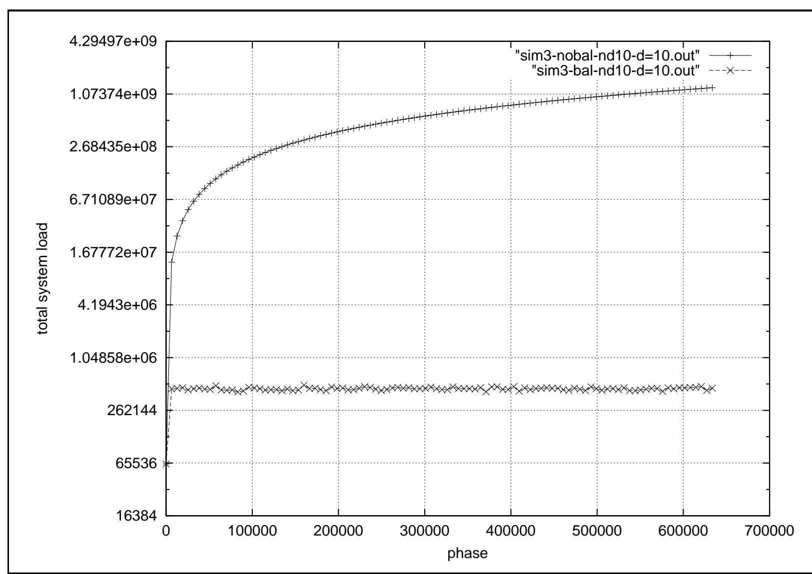

Figure 6.20: Adversarial generation, algorithm ALGADV, $n/10$ generators, $\Delta = 10$ (logarithmic scale on y-axis)

CHAPTER 7

Conclusions and Outlook

We have presented three load balancing algorithms, two for the stochastical generation model, and one for the adversarial one. The algorithms are quite simple in concept, but still might seem to be rather... theoretical. Since the collision game is the basic building block of all of them, one shouldn't even think of actually implementing them if it is not absolutely sure that the collision game can be implemented very efficiently. As we have seen in the simulations, the algorithms are quite robust as far as hand tuning of almost all parameters is concerned, the thresholds as well as the parameters for the collision games themselves.

Still, the algorithms have another merit in their own right. Compared to the method of distributing tasks using a balls into bins approach, we clearly save on communication. This is mainly due to the fact that we transfer tasks only if a certain load threshold is exceeded, while in balls into bins games each and every task is thrown away upon generation. Compared to other approaches, we could show the first high probability bound on the maximum load and the difference between the maximum load and the average system load, respectively (other algorithms needed the system load to be sufficiently large in order to obtain similar bounds).

Bibliography

[ABKU94] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations (extended abstract). In *Proceedings of the 26th Symposium on Theory of Computing (STOC'94)*, pages 593–602, 1994.

[ABS98] Micah Adler, Petra Berenbrink, and Klaus Schröder. Analyzing an infinite parallel job allocation process. In *Proceedings of the 6th European Symposium on Algorithms (ESA'98)*, pages 417–428, 1998.

[ACMR95] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. Parallel randomized load balancing. In *Proceedings of the 27th Symposium on Theory of Computing*, pages 238–247, New York, NY, USA, May 29–June 1 1995. ACM Press.

[BCFV99] Petra Berenbrink, Artur Czumaj, Tom Friedetzky, and Nikita Vvedenskaya. On the analysis of infinite parallel job allocation processes via differential equations. Manuscript, 1999.

[BCSV00] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: The heavily loaded case. In *Proceedings of the 32th ACM Symposium on Theory of Computing (STOC'00)*, 2000.

[BDG+99] Michael Booth, Jim Davies, Mark Galassi, Brian Gough, Gerard Jungman, Reid Priedhorsky, and James Theiler. GNU scientific library (GSL), version 0.5. http://www.cygnus.com/gsl/, 1999.

[Ber00] Petra Berenbrink. *Randomized Allocation of Independent Tasks*. PhD thesis, University of Paderborn, 2000.

[BFM98] Petra Berenbrink, Tom Friedetzky, and Ernst W. Mayr. Parallel continuous randomized load balancing. In *Proceedings of the 10th Symposium on Parallel Algorithms and Architectures (SPAA '98)*, pages 192–201. ACM, 1998.

[BFS99] Petra Berenbrink, Tom Friedetzky, and Angelika Steger. Randomized and adversarial load balancing. In *Proceedings of the 11th Symposium on Parallel Algorithms and Architectures (SPAA'99)*, pages 175–184, 1999.

[BL94] Robert Blumofe and Charles Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 37th Symphosium on Foundations of Computer Science (FOCS'94)*, pages 362–371, 1994.

[BMS97] Petra Berenbrink, Friedhelm Meyer auf der Heide, and Klaus Schröder. Allocating weighted jobs in parallel. In *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures (SPAA '97)*, pages 302–310, 1997.

[Boi90] J.E. Boillat. Load balancing and poisson equation in a graph. *Concurrency: Practice and Experience*, 2(4):289–313, 1990.

[CFM+98] Richard Cole, Alan Frieze, Bruce M. Maggs, Michael Mitzenmacher, Andrea W. Richa, Ramesh K. Sitaraman, and Eli Upfal. On balls and bins with delitions. In M. Luby, J. Rolim, and M. Serna, editors, *Proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 145–158, 1998.

[Coh82] Jacob W. Cohen. *The Single Server Queue.* North-Holland Publ. Co, 1982.

[CS97] Artur Czumaj and Volker Stemann. Randomized allocation processes. In *Proceedings of the 38th Symposium on Foundations on Computer Science (FOCS'97)*, pages 194–203, 1997.

[Cyb89] G. Cybenko. Load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, pages 279–301, 1989.

[Czu98] Artur Czumaj. Recovery time of dynamic allocation processes. In *Proceedings of the 10th Symposium on Parallel Algorithms and Architectures (SPAA'98)*, pages 202–211, 1998.

[DHB97] S. K. Das, D. J. Harvey, and R. Biswas. Design of novel load balancing algorithms with implementations on an ibm sp2. In *Proc. of the 3rd EURO-PAR Conference*, 1997.

[DM90] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. How to distribute a dictionary in a complete network. In *Proceedings of the 22nd Symposium on Theory of Computing (STOC'90)*, pages 117–127, 1990.

[GLM+99] Bhaskar Ghosh, Thomas Leighton, Bruce Maggs, S. Muthukrishnan, C.G. Plaxton, Rajmohan Rajaraman, Andrea Richa, Robert Tarjan, and David Zuckerman. Tight analyses of two local load balancing algorithms. *SIAM Journal of Computing*, 29:29–64, 1999.

[GMS96] Bhaskar Ghosh, S. Muthukrishnan, and M.H. Schultz. First and second order diffusive methods for rapid, coarse, distributed load balancing. In *Proceedings of the 8th Symposium on Parallel Algorithms and Architectures(SPAA'96)*, pages 72–81, 1996.

[Hof87] M. Hofri. *Probabilistic Analysis of Algorithms*. Springer Verlag, New York, 1987.

[HR89] Torben Hagerub and Christine Rüb. A guided tour of chernov bounds. *Information Processing Letters 33*, pages 305–308, 1989.

[HS97] C.-J. Hou and K. G. Shin. Implementation of decentralized load sharing in networked workstations using the Condor package. *Journal of Parallel and Distributed Computing*, 40:173–184, 1997.

[Kin62] J. F. C. Kingman. The effect of queue discipline on waiting time variance. *Proceedings of Chambridge University Press*, 1962.

[Kin64] J. F. C. Kingman. A martingale inequality in the theory of queues. *Proceedings of Chambridge University Press*, 60:359–361, 1964.

[Kle96] Leonard Kleinrock. *Queueing systems: Problems and Solutions.* Wiley, 1996.

[KLM92] Richard Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient PRAM simulations on a distributed memory machine. In *Proceedings of the 24th Symposium on Theory of Computing (STOC'92)*, pages 318–326, 1992.

[KZ88] Richard Karp and Yanjun Zhang. A randomized branch-and-bound procedure. In *Proceedings of the 20th Symposium on Theory of Computing (STOC'88)*, pages 290–300, 1988.

[Lau95] Thomas Lauer. *Adaptive Dynamische Lastbalancierung.* PhD thesis, Universität des Saarlandes, 1995.

[LM93] Reinhard Lüling and Burkhard Monien. A dynamic distributed load balancing algorithm with provable good performance. In *Proceedings of the 5th Symposium on Parallel Algorithms and Architectures*, pages 164–172. ACM Press, 1993.

[Lül96] Reinhard Lüling. *Lastverteilungsverfahren zur effizienten Nutzung paralleler Systeme.* PhD thesis, University of Paderborn, 1996.

[MD96] N. R. Mahapatra and S. Dutt. Random seeking: A general, efficient, and informed randomized scheme for dynamic load balancing. In *Proc. of the 10th International Parallel Processing Symposium (IPPS'96)*, pages pages 881–885, 1996.

[Mit96a] M. Mitzenmacher. Load balancing and density dependent jump markov processes. In *Proceedings of 37th Symposium on Foundations of Computer Science*, pages 213–223. IEEE, Oct 1996.

[Mit96b] Michael Mitzenmacher. Density dependent jump markov processes and applications to load balancing. In *Proceedings of the 37th Symposium on Foundations of Computer Science (FOCS'96)*, pages 213–222, 1996.

[Mit96c] Michael Mitzenmacher. *The Power of Two Random Choices in Randomized Load Balancing.* PhD thesis, Graduate Division of the University of California at Berkley, 1996.

[Mit97] Michael Mitzenmacher. On the analysis of randomized load balancing schemes. In *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures (SPAA'97)*, pages 292–301, 1997.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, january 1998.

[MR98] S. Muthukrishnan and Rajmohan Rajaraman. An adversarial model for dynamic load balancing. In *Proceedings of the 10th Symposium on Parallel Algorithms and Architectures (SPAA'98)*, pages 47–54, 1998.

[MSS95] Friedhelm Meyer auf der Heide, Christian Scheideler, and Volker Stemann. Exploiting storage redundancy to speed up randomized shared memory simulations. In *Proceedings of the 12th Symposium on Theoretical Aspects of Computer Science (Stacs'95)*, pages 267–278, 1995.

[RS98] Martin Raab and Angelika Steger. Balls into bins — a simple and tight analysis. In *Proceedings of the second International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '98)*, pages 159–170, 1998.

[RSU91] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the 3rd Symposium on Parallel Algorithms and Architectures*, pages 237–245. ACM Press, 1991.

[SG97] Thomas Schnekenburger and G.Stellner. *Dynamic Load Distribution for Parallel Applications.* B.G. Teubener Verlagsgesellschaft, 1997.

[SS97] T. Schnekenburger and G. Stellner. *Dynamic Load Distribution for Parallel Applications.* Teubner, 1997.

[Ste96a] Volker Stemann. Parallel balanced allocations. In *Proceedings of the 8th Symposium on Parallel Algorithms and Architectures (SPAA '96)*, pages 261–269. ACM, 1996.

[Ste96b] Volker Stemann. Parallel balanced allocations (extended abstract). In *Proceedings of the 8th Symposium on Parallel Algorithms and Architectures (SPAA '96)*, pages 261–269, 1996.

[Tri92] Kishor S. Trivedi. *Probability & statistics with reliability, queueing, and compute science applications.* Prentice-Hall, 1992.

[VDK96] N. D. Vvedenskaya, R. L. Dobrushin, and F. I. Karpelevich. Queueing system with selection of the shortest of two queues: an asymptotic approach. *Problems of Information Transmission*, 32(1):15–27, 1996.

[Vöc00] Berthold Vöcking. How asymmetry helps load balancing. In *Proceedings of the 40th Symphosium on Foundations of Computer Science (FOCS'00)*, pages 131–140, 2000.

[VS97] Nikita Vvedenskaya and Yuri Suhov. Dobrushin's mean-field approximation for queue with dynamic routing. Technical report, INRIA Rocquencourt, France, 1997.

[WHV95] G. S. Wolffe, S. H. Hosseini, and K. Vairavan. Performance of an adaptive algorithm for dynamic load balancing. In *Proc. of the ISCA International Conference on Parallel and Distributed Computing Systems*, pages 613–618, 1995.